Broadview
www.broadview.com.cn

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, world or 你好，世界 or καλημέρα κόσμε or こんにちは世界\n")
}
```

Go Web 编程

谢孟军 著



Go语言性能好、语法简单、开发效率高！
一起来探索Go语言吧！

电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

目錄

Go Web 编程

作者：astaxie

来源：build-web-application-with-golang

Donate

AliPay:

English Donate:donate

Community

QQ群：148647580

BBS：http://golanghome.com/

Acknowledgments

- 四月份平民 (review代码)
- Hong Ruiqi (review代码)
- BianJiang (编写go开发工具Vim和Emacs的设置)
- Oling Cat(review代码)
- Wenlei Wu(提供一些图片展示)
- polaris(review书)
- 雨痕(review第二章)

License

1 GO环境配置

欢迎来到Go的世界，让我们开始探索吧！

Go是一种新的语言，一种并发的、带垃圾回收的、快速编译的语言。它具有以下特点：

- 它可以在一台计算机上用几秒钟的时间编译一个大型的Go程序。
- Go为软件构造提供了一种模型，它使依赖分析更加容易，且避免了大部分C风格include文件与库的开头。
- Go是静态类型的语言，它的类型系统没有层级。因此用户不需要在定义类型之间的关系上花费时间，这样感觉起来比典型的面向对象语言更轻量级。
- Go完全是垃圾回收型的语言，并为并发执行与通信提供了基本的支持。
- 按照其设计，Go打算为多核机器上系统软件的构造提供一种方法。

Go是一种编译型语言，它结合了解释型语言的游刃有余，动态类型语言的开发效率，以及静态类型的安全性。它也打算成为现代的，支持网络与多核计算的语言。要满足这些目标，需要解决一些语言上的问题：一个富有表达能力但轻量级的类型系统，并发与垃圾回收机制，严格的依赖规范等等。这些无法通过库或工具解决好，因此Go也就应运而生了。

在本章中，我们将讲述Go的安装方法，以及如何配置项目信息。

目录

links

- 目录
- 下一节: Go安装

1.1 Go 安装

Go的三种安装方式

Go有多种安装方式，你可以选择自己喜欢的。这里我们介绍三种最常见的安装方式：

- Go源码安装：这是一种标准的软件安装方式。对于经常使用Unix类系统的用户，尤其对于开发者来说，从源码安装可以自己定制。
- Go标准包安装：Go提供了方便的安装包，支持Windows、Linux、Mac等系统。这种方式适合快速安装，可根据自己的系统位数下载好相应的安装包，一路next就可以轻松安装了。推荐这种方式
- 第三方工具安装：目前有很多方便的第三方软件包工具，例如Ubuntu的apt-get、Mac的homebrew等。这种安装方式适合那些熟悉相应系统的用户。

最后，如果你想在同一个系统中安装多个版本的Go，你可以参考第三方工具GVM，这是目前在这方面做得最好的工具，除非你知道怎么处理。

Go源码安装

在Go的源代码中，有些部分是用Plan 9 C和AT&T汇编写的，因此假如你要想从源码安装，就必须安装C的编译工具。

在Mac系统中，只要你安装了Xcode，就已经包含了相应的编译工具。

在类Unix系统中，需要安装gcc等工具。例如Ubuntu系统可通过在终端中执行 `sudo apt-get install gcc libc6-dev` 来安装编译工具。

在Windows系统中，你需要安装MinGW，然后通过MinGW安装gcc，并设置相应的环境变量。

你可以直接去官网下载源码，找相应的 `goVERSION.src.tar.gz` 的文件下载，下载之后解压缩到 `$HOME` 目录，执行如下代码：

```
cd go/src
./all.bash
```

运行all.bash后出现"ALL TESTS PASSED"字样时才算安装成功。

上面是Unix风格的命令，Windows下的安装方式类似，只不过是运行 `all.bat` ，调用的编译器是MinGW的gcc。

如果是Mac或者Unix用户需要设置几个环境变量，如果想重启之后也能生效的话把下面的命令写到 `.bashrc` 或者 `.zshrc` 里面，

```
export GOPATH=$HOME/gopath
export PATH=$PATH:$HOME/go/bin:$GOPATH/bin
```

如果你是写入文件的，记得执行 `bash .bashrc` 或者 `bash .zshrc` 使得设置立马生效。

如果是window系统，就需要设置环境变量，在path里面增加相应的go所在的目录，设置gopath变量。

当你设置完毕之后在命令行里面输入 `go` ，看到如下图片即说明你已经安装成功

图1.1 源码安装之后执行Go命令的图

如果出现Go的Usage信息，那么说明Go已经安装成功了；如果出现该命令不存在，那么可以检查一下自己的PATH环境变中是否包含了Go的安装目录。

> 关于上面的GOPATH将在下面小节详细讲解

Go标准包安装

Go提供了每个平台打好包的一键安装，这些包默认会安装到如下目录：/usr/local/go (Windows系统：c:\Go)，当然你可以改变他们的安装位置，但是改变之后你必须在你的环境变量中设置如下信息：

```
export GOROOT=$HOME/go
export GOPATH=$HOME/gopath
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

上面这些命令对于Mac和Unix用户来说最好是写入 `.bashrc` 或者 `.zshrc` 文件，对于windows用户来说当然是写入环境变量。

如何判断自己的操作系统是32位还是64位？

我们接下来的Go安装需要判断操作系统的位数，所以这小节我们先确定自己的系统类型。

Windows系统用户请按Win+R运行cmd，输入 `systeminfo` 后回车，稍等片刻，会出现一些系统信息。在“系统类型”一行中，若显示“x64-based PC”，即为64位系统；若显示“X86-based PC”，则为32位系统。

Mac系统用户建议直接使用64位的，因为Go所支持的Mac OS X版本已经不支持纯32位处理器了。

Linux系统用户可通过在Terminal中执行命令 `arch` (即 `uname -m`)来查看系统信息：

64位系统显示

```
x86_64
```

32位系统显示

```
i386
```

Mac 安装

访问下载地址，32位系统下载go1.4.2.darwin-386-osx10.8.pkg，64位系统下载go1.4.2.darwin-amd64-osx10.8.pkg，双击下载文件，一路默认安装点击下一步，这个时候go已经安装到你的系统中，默认已经在PATH中增加了相应的 `~/go/bin` ，这个时候打开终端，输入 `go`

看到类似上面源码安装成功的图片说明已经安装成功

如果出现go的Usage信息，那么说明go已经安装成功了；如果出现该命令不存在，那么可以检查一下自己的PATH环境变中是否包含了go的安装目录。

Linux 安装

访问下载地址，32位系统下载go1.4.2.linux-386.tar.gz，64位系统下载go1.4.2.linux-amd64.tar.gz，

假定你想要安装Go的目录为 `$GO_INSTALL_DIR` ，后面替换为相应的目录路径。

解压缩 `tar.gz` 包到安装目录下： `tar zxvf go1.4.2.linux-amd64.tar.gz -C $GO_INSTALL_DIR` 。

设置PATH， `export PATH=$PATH:$GO_INSTALL_DIR/go/bin`

然后执行 `go`

图1.2 Linux系统下安装成功之后执行go显示的信息

如果出现go的Usage信息，那么说明go已经安装成功了；如果出现该命令不存在，那么可以检查一下自己的PATH环境变中是否包含了go的安装目录。

Windows 安装

访问Google Code 下载页，32 位请选择名称中包含 windows-386 的 msi 安装包，64 位请选择名称中包含 windows-amd64 的。下载好后运行，不要修改默认安装目录 C:\Go\，若安装到其他位置会导致不能执行自己所编写的 Go 代码。安装完成后

默认会在环境变量 Path 后添加 Go 安装目录下的 bin 目录 `C:\Go\bin\` ，并添加环境变量 GOROOT，值为 Go 安装根目录 `C:\Go\` 。

验证是否安装成功

在运行中输入 `cmd` 打开命令行工具，在提示符下输入 `go` ，检查是否能看到 Usage 信息。输入 `cd %GOROOT%` ，看是否能进入 Go 安装目录。若都成功，说明安装成功。

不能的话请检查上述环境变量 Path 和 GOROOT 的值。若不存在请卸载后重新安装，存在请重启计算机后重试以上步骤。

第三方工具安装

GVM

gvm是第三方开发的Go多版本管理工具，类似ruby里面的rvm工具。使用起来相当的方便，安装gvm使用如下命令：

```
bash < <(curl -s -S -L https://raw.githubusercontent.com/moovweb/gv
```

安装完成后我们就可以安装go了：

```
gvm install go1.4.2
gvm use go1.4.2
```

也可以使用下面的命令，省去每次调用gvm use的麻烦： gvm use go1.4.2 --default

执行完上面的命令之后GOPATH、GOROOT等环境变量会自动设置好，这样就可以直接使用了。

apt-get

Ubuntu是目前使用最多的Linux桌面系统，使用 `apt-get` 命令来管理软件包，我们可以通过下面的命令来安装Go，为了以后方便，应该把 `git` `mercurial` 也安装上：

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:gophers/go
sudo apt-get update
sudo apt-get install golang-stable git-core mercurial
```

homebrew

homebrew是Mac系统下面目前使用最多的管理软件的工具，目前已支持Go，可以通过命令直接安装Go，为了以后方便，应该把 `git` `mercurial` 也安装上：

```
brew update && brew upgrade
brew install go
brew install git
brew install mercurial
```

links

- 目录
- 上一节: Go环境配置
- 下一节: GOPATH 与工作空间

1.2 GOPATH与工作空间

前面我们在安装Go的时候看到需要设置GOPATH变量，Go从1.1版本开始必须设置这个变量，而且不能和Go的安装目录一样，这个目录用来存放Go源码，Go的可运行文件，以及相应的编译之后的包文件。所以这个目录下面有三个子目录：src、bin、pkg

GOPATH设置

go 命令依赖一个重要的环境变量：$GOPATH

Windows系统中环境变量的形式为 `%GOPATH%` ，本书主要使用Unix形式，Windows用户请自行替换。

（注：这个不是*Go*安装目录。下面以笔者的工作目录为示例，如果你想不一样请把*GOPATH*替换成你的工作目录。）

在类似 Unix 环境大概这样设置：

```
export GOPATH=/home/apple/mygo
```

为了方便，应该新建以上文件夹，并且上一行加入到 `.bashrc` 或者 `.zshrc` 或者自己的 `sh` 的配置文件中。

Windows 设置如下，新建一个环境变量名称叫做GOPATH：

```
    GOPATH=c:\mygo
```

GOPATH允许多个目录，当有多个目录时，请注意分隔符，多个目录的时候Windows是分号，Linux系统是冒号，当有多个GOPATH时，默认会将go get的内容放在第一个目录下。

以上 $GOPATH 目录约定有三个子目录：

- src 存放源代码（比如：.go .c .h .s等）
- pkg 编译后生成的文件（比如：.a）
- bin 编译后生成的可执行文件（为了方便，可以把此目录加入到 $PATH 变量中，如果有多个gopath，那么使用 `${GOPATH//://bin:}/bin` 添加所有的bin目录）

以后我所有的例子都是以mygo作为我的gopath目录

代码目录结构规划

GOPATH下的src目录就是接下来开发程序的主要目录，所有的源码都是放在这个目录下面，那么一般我们的做法就是一个目录一个项目，例如: $GOPATH/src/mymath 表示mymath这个应用包或者可执行应用，这个根据package是main还是其他来决定，main的话就是可执行应用，其他的话就是应用包，这个会在后续详细介绍package。

所以当新建应用或者一个代码包时都是在src目录下新建一个文件夹，文件夹名称一般是代码包名称，当然也允许多级目录，例如在src下面新建了目录$GOPATH/src/github.com/astaxie/beedb 那么这个包路径就是"github.com/astaxie/beedb"，包名称是最后一个目录beedb

下面我就以mymath为例来讲述如何编写应用包，执行如下代码

```
cd $GOPATH/src
mkdir mymath
```

新建文件sqrt.go，内容如下

```
// $GOPATH/src/mymath/sqrt.go源码如下：
package mymath

func Sqrt(x float64) float64 {
    z := 0.0
    for i := 0; i < 1000; i++ {
        z -= (z*z - x) / (2 * x)
    }
    return z
}
```

这样我的应用包目录和代码已经新建完毕，注意：一般建议package的名称和目录名保持一致

编译应用

上面我们已经建立了自己的应用包，如何进行编译安装呢？有两种方式可以进行安装

1、只要进入对应的应用包目录，然后执行 `go install` ，就可以安装了

2、在任意的目录执行如下代码 `go install mymath`

安装完之后，我们可以进入如下目录

```
cd $GOPATH/pkg/${GOOS}_${GOARCH}
//可以看到如下文件
mymath.a
```

这个.a文件是应用包，那么我们如何进行调用呢？

接下来我们新建一个应用程序来调用这个应用包

新建应用包mathapp

```
cd $GOPATH/src
mkdir mathapp
cd mathapp
vim main.go
```

`$GOPATH/src/mathapp/main.go` 源码：

```
package main

import (
	"mymath"
	"fmt"
)

func main() {
	fmt.Printf("Hello, world.  Sqrt(2) = %v\n", mymath.Sqrt(2))
}
```

可以看到这个的package是 `main` ，import里面调用的包是 `mymath` ,这个就是相对于 `$GOPATH/src` 的路径，如果是多级目录，就在import里面引入多级目录，如果你有多个GOPATH，也是一样，Go会自动在多个 `$GOPATH/src` 中寻找。

如何编译程序呢？进入该应用目录，然后执行 `go build` ，那么在该目录下面会生成一个mathapp的可执行文件

```
./mathapp
```

输出如下内容

```
Hello, world.  Sqrt(2) = 1.414213562373095
```

如何安装该应用，进入该目录执行 `go install` ,那么在$GOPATH/bin/下增加了一个可执行文件mathapp, 还记得前面我们把 `$GOPATH/bin` 加到我们的PATH里面了，这样可以在命令行输入如下命令就可以执行

```
mathapp
```

也是输出如下内容

```
Hello, world.  Sqrt(2) = 1.414213562373095
```

这里我们展示如何编译和安装一个可运行的应用，以及如何设计我们的目录结构。

获取远程包

go语言有一个获取远程包的工具就是 `go get` ，目前go get支持多数开源社区(例如：github、googlecode、bitbucket、Launchpad)

```
go get github.com/astaxie/beedb
```

> go get -u 参数可以自动更新包，而且当go get的时候会自动获取该包依赖的其他第三方包

通过这个命令可以获取相应的源码，对应的开源平台采用不同的源码控制工具，例如github采用git、googlecode采用hg，所以要想获取这些源码，必须先安装相应的源码控制工具

通过上面获取的代码在我们本地的源码相应的代码结构如下

```
$GOPATH
  src
   |--github.com
          |-astaxie
              |-beedb
   pkg
    |--相应平台
         |-github.com
               |--astaxie
                    |beedb.a
```

go get本质上可以理解为首先第一步是通过源码工具clone代码到src下面，然后执行 `go install`

在代码中如何使用远程包，很简单的就是和使用本地包一样，只要在开头import相应的路径就可以

```
import "github.com/astaxie/beedb"
```

程序的整体结构

通过上面建立的我本地的mygo的目录结构如下所示

```
bin/
    mathapp
pkg/
    平台名/ 如：darwin_amd64、linux_amd64
         mymath.a
         github.com/
              astaxie/
                   beedb.a
src/
    mathapp
          main.go
      mymath/
          sqrt.go
      github.com/
           astaxie/
                beedb/
                    beedb.go
                    util.go
```

从上面的结构我们可以很清晰的看到，bin目录下面存的是编译之后可执行的文件，pkg下面存放的是应用包，src下面保存的是应用源代码

links

- 目录
- 上一节: GO安装
- 下一节: GO 命令

1.3 Go 命令

Go 命令

Go语言自带有一套完整的命令操作工具，你可以通过在命令行中执行 `go` 来查看它们：

图1.3 Go命令显示详细的信息

这些命令对于我们平时编写的代码非常有用，接下来就让我们了解一些常用的命令。

go build

这个命令主要用于编译代码。在包的编译过程中，若有必要，会同时编译与之相关联的包。

- 如果是普通包，就像我们在1.2节中编写的 `mymath` 包那样，当你执行 `go build` 之后，它不会产生任何文件。如果你需要在 `$GOPATH/pkg` 下生成相应的文件，那就得执行 `go install` 。
- 如果是 `main` 包，当你执行 `go build` 之后，它就会在当前目录下生成一个可执行文件。如果你需要在 `$GOPATH/bin` 下生成相应的文件，需要执行 `go install` ，或者使用 `go build -o 路径/a.exe` 。
- 如果某个项目文件夹下有多个文件，而你只想编译某个文件，就可在 `go build` 之后加上文件名，例如 `go build a.go` ； `go build` 命令默认会编译当前目录下的所有go文件。
- 你也可以指定编译输出的文件名。例如1.2节中的 `mathapp` 应用，我们可以指定 `go build -o astaxie.exe` ，默认情况是你的package名(非main包)，或者是第一个源文件的文件名(main包)。

  （注：实际上，package名在Go语言规范中指代码中“package”后使用的名称，此名称可以与文件夹名不同。默认生成的可执行文件名是文件夹名。）
- go build会忽略目录下以“_”或“.”开头的go文件。
- 如果你的源代码针对不同的操作系统需要不同的处理，那么你可以根据不同的操作系统后缀来命名文件。例如有一个读取数组的程序，它对于不同的操作系统可能有如下几个源文件：

  array_linux.go array_darwin.go array_windows.go array_freebsd.go

`go build`的时候会选择性地编译以系统名结尾的文件（Linux、Darwin、Windows、Freebsd）。例如Linux系统下面编译只会选择array_linux.go文件，其它系统命名后缀文件全部忽略。

参数的介绍

- `-o` 指定输出的文件名，可以带上路径，例如 `go build -o a/b/c`
- `-i` 安装相应的包，编译+ `go install`
- `-a` 更新全部已经是最新的包的，但是对标准包不适用
- `-n` 把需要执行的编译命令打印出来，但是不执行，这样就可以很容易的知道底层是如何运行的
- `-p n` 指定可以并行可运行的编译数目，默认是CPU数目
- `-race` 开启编译的时候自动检测数据竞争的情况，目前只支持64位的机器
- `-v` 打印出来我们正在编译的包名
- `-work` 打印出来编译时候的临时文件夹名称，并且如果已经存在的话就不要删除
- `-x` 打印出来执行的命令，其实就是和 `-n` 的结果类似，只是这个会执行
- `-ccflags 'arg list'` 传递参数给5c, 6c, 8c 调用
- `-compiler name` 指定相应的编译器，gccgo还是gc
- `-gccgoflags 'arg list'` 传递参数给gccgo编译连接调用
- `-gcflags 'arg list'` 传递参数给5g, 6g, 8g 调用
- `-installsuffix suffix` 为了和默认的安装包区别开来，采用这个前缀来重新安装那些依赖的包，`-race` 的时候默认已经是 `-installsuffix race`，大家可以通过 `-n` 命令来验证
- `-ldflags 'flag list'` 传递参数给5l, 6l, 8l 调用
- `-tags 'tag list'` 设置在编译的时候可以适配的那些tag，详细的tag限制参考里面的 Build Constraints

go clean

这个命令是用来移除当前源码包和关联源码包里面编译生成的文件。这些文件包括

```
_obj/            旧的object目录，由Makefiles遗留
_test/           旧的test目录，由Makefiles遗留
_testmain.go     旧的gotest文件，由Makefiles遗留
test.out         旧的test记录，由Makefiles遗留
build.out        旧的test记录，由Makefiles遗留
*.[568ao]        object文件，由Makefiles遗留

DIR(.exe)        由go build产生
DIR.test(.exe)   由go test -c产生
MAINFILE(.exe)   由go build MAINFILE.go产生
*.so             由 SWIG 产生
```

我一般都是利用这个命令清除编译文件，然后github递交源码，在本机测试的时候这些编译文件都是和系统相关的，但是对于源码管理来说没必要。

```
$ go clean -i -n
cd /Users/astaxie/develop/gopath/src/mathapp
rm -f mathapp mathapp.exe mathapp.test mathapp.test.exe app app.exe
rm -f /Users/astaxie/develop/gopath/bin/mathapp
```

参数介绍

- `-i` 清除关联的安装的包和可运行文件，也就是通过go install安装的文件
- `-n` 把需要执行的清除命令打印出来，但是不执行，这样就可以很容易的知道底层是如何运行的
- `-r` 循环的清除在import中引入的包
- `-x` 打印出来执行的详细命令，其实就是 `-n` 打印的执行版本

go fmt

有过C/C++经验的读者会知道,一些人经常为代码采取K&R风格还是ANSI风格而争论不休。在go中，代码则有标准的风格。由于之前已经有的一些习惯或其它的原因我们常将代码写成ANSI风格或者其它更合适自己的格式，这将为人们在阅读别人的代码时添加不必要的负担，所以go强制了代码格式（比如左大括号必须放在行尾），不按照此格式的代码将不能编译通过，为了减少浪费在排版上的时间，go工具集中提供了一个 `go fmt` 命令 它可以帮你格式化你写好的代码文件，使你写代码的时候不需要关心格式，你只需要在写完之后执行 `go fmt <文件名>.go` ，你的代码就被修改成了标准格式，但是我平常很少用到这个命令，因为开发工具里面一般都带了保存时候自动格式化功能，这个功能其实在底层就是调用了 `go fmt` 。接下来的一节我将讲述两个工具，这两个工具都自带了保存文件时自动化 `go fmt` 功能。

使用go fmt命令，其实是调用了gofmt，而且需要参数-w，否则格式化结果不会写入文件。gofmt -w -l src，可以格式化整个项目。

所以go fmt是gofmt的上层一个包装的命令，我们想要更多的个性化的格式化可以参考 gofmt

gofmt的参数介绍

- `-l` 显示那些需要格式化的文件
- `-w` 把改写后的内容直接写入到文件中，而不是作为结果打印到标准输出。
- `-r` 添加形如“a[b:len(a)] -> a[b:]”的重写规则，方便我们做批量替换
- `-s` 简化文件中的代码
- `-d` 显示格式化前后的diff而不是写入文件，默认是false
- `-e` 打印所有的语法错误到标准输出。如果不使用此标记，则只会打印不同行的前10个错误。
- `-cpuprofile` 支持调试模式，写入相应的cpufile到指定的文件

go get

这个命令是用来动态获取远程代码包的，目前支持的有BitBucket、GitHub、Google Code和Launchpad。这个命令在内部实际上分成了两步操作：第一步是下载源码包，第二步是执行 `go install` 。下载源码包的go工具会自动根据不同的域名调用不同的源码工具，对应关系如下：

```
BitBucket (Mercurial Git)
GitHub (Git)
Google Code Project Hosting (Git, Mercurial, Subversion)
Launchpad (Bazaar)
```

所以为了 `go get` 能正常工作，你必须确保安装了合适的源码管理工具，并同时把这些命令加入你的PATH中。其实 `go get` 支持自定义域名的功能，具体参见 `go help remote` 。

参数介绍：

- `-d` 只下载不安装
- `-f` 只有在你包含了 `-u` 参数的时候才有效，不让 `-u` 去验证import中的每一个都已经获取了，这对于本地fork的包特别有用
- `-fix` 在获取源码之后先运行fix，然后再去做其他的事情
- `-t` 同时也下载需要为运行测试所需要的包
- `-u` 强制使用网络去更新包和它的依赖包
- `-v` 显示执行的命令

go install

这个命令在内部实际上分成了两步操作：第一步是生成结果文件(可执行文件或者.a包)，第二步会把编译好的结果移到 `$GOPATH/pkg` 或者 `$GOPATH/bin` 。

参数支持 `go build` 的编译参数。大家只要记住一个参数 `-v` 就好了，这个随时随地的可以查看底层的执行信息。

go test

执行这个命令，会自动读取源码目录下面名为 `*_test.go` 的文件，生成并运行测试用的可执行文件。输出的信息类似

```
ok   archive/tar   0.011s
FAIL archive/zip   0.022s
ok   compress/gzip 0.033s
...
```

默认的情况下，不需要任何的参数，它会自动把你源码包下面所有test文件测试完毕，当然你也可以带上参数，详情请参考 `go help testflag`

这里我介绍几个我们常用的参数：

- `-bench regexp` 执行相应的benchmarks，例如 `-bench=.`
- `-cover` 开启测试覆盖率
- `-run regexp` 只运行regexp匹配的函数，例如 `-run=Array` 那么就执行包含有Array开头的函数
- `-v` 显示测试的详细命令

go tool

`go tool` 下面下载聚集了很多命令，这里我们只介绍两个，fix和vet

- `go tool fix .` 用来修复以前老版本的代码到新版本，例如go1之前老版本的代码转化到go1,例如API的变化
- `go tool vet directory|files` 用来分析当前目录的代码是否都是正确的代码,例如是不是调用fmt.Printf里面的参数不正确，例如函数里面提前return了然后出现了无用代码之类的。

go generate

这个命令是从Go1.4开始才设计的，用于在编译前自动化生成某类代码。 `go generate` 和 `go build` 是完全不一样的命令，通过分析源码中特殊的注释，然后执行相应的命令。这些命令都是很明确的，没有任何的依赖在里面。而且大家在用这个之前心里面一定要有一个理念，这个 `go generate` 是给你用的，不是给使用你这个包的人用的，是方便你来生成一些代码的。

这里我们来举一个简单的例子，例如我们经常会使用 `yacc` 来生成代码，那么我们常用这样的命令：

```
go tool yacc -o gopher.go -p parser gopher.y
```

-o 指定了输出的文件名， -p指定了package的名称，这是一个单独的命令，如果我们想让 `go generate` 来触发这个命令，那么就可以在当然目录的任意一个 `xxx.go` 文件里面的任意位置增加一行如下的注释：

```
//go:generate go tool yacc -o gopher.go -p parser gopher.y
```

这里我们注意了， `//go:generate` 是没有任何空格的，这其实就是一个固定的格式，在扫描源码文件的时候就是根据这个来判断的。

所以我们可以通过如下的命令来生成，编译，测试。如果 `gopher.y` 文件有修改，那么就重新执行 `go generate` 重新生成文件就好。

```
$ go generate
$ go build
$ go test
```

godoc

在Go1.2版本之前还支持 `go doc` 命令，但是之后全部已到了godoc这个命令下，需要这样安装 `go get golang.org/x/tools/cmd/godoc`

很多人说go不需要任何的第三方文档，例如chm手册之类的（其实我已经做了一个了，chm手册），因为它内部就有一个很强大的文档工具。

如何查看相应package的文档呢？例如builtin包，那么执行 `godoc builtin` 如果是http包，那么执行 `godoc net/http` 查看某一个包里面的函数，那么执行 `godoc fmt Printf` 也可以查看相应的代码，执行 `godoc -src fmt Printf`

通过命令在命令行执行 godoc -http=:端口号 比如 `godoc -http=:8080` 。然后在浏览器中打开 `127.0.0.1:8080` ，你将会看到一个golang.org的本地copy版本，通过它你可以查询pkg文档等其它内容。如果你设置了GOPATH，在pkg分类下，不但会列出标准包的文档，还会列出你本地 `GOPATH` 中所有项目的相关文档，这对于经常被墙的用户来说是一个不错的选择。

其它命令

go还提供了其它很多的工具，例如下面的这些工具

```
go version 查看go当前的版本
go env 查看当前go的环境变量
go list 列出当前全部安装的package
go run 编译并运行Go程序
```

以上这些工具还有很多参数没有一一介绍，用户可以使用 `go help 命令` 获取更详细的帮助信息。

links

- 目录
- 上一节: GOPATH与工作空间
- 下一节: Go开发工具

1.4 Go开发工具

本节我将介绍几个开发工具，它们都具有自动化提示，自动化fmt功能。因为它们都是跨平台的，所以安装步骤之类的都是通用的。

LiteIDE

LiteIDE是一款专门为Go语言开发的跨平台轻量级集成开发环境（IDE），由visualfc编写。

图1.4 LiteIDE主界面

LiteIDE主要特点：

- 支持主流操作系统
    - Windows
    - Linux
    - MacOS X
- Go编译环境管理和切换
    - 管理和切换多个Go编译环境
    - 支持Go语言交叉编译
- 与Go标准一致的项目管理方式
    - 基于GOPATH的包浏览器
    - 基于GOPATH的编译系统
    - 基于GOPATH的Api文档检索
- Go语言的编辑支持
    - 类浏览器和大纲显示
    - Gocode(代码自动完成工具)的完美支持
    - Go语言文档查看和Api快速检索
    - 代码表达式信息显示 `F1`
    - 源代码定义跳转支持 `F2`
    - Gdb断点和调试支持
    - gofmt自动格式化支持
- 其他特征
    - 支持多国语言界面显示
    - 完全插件体系结构
    - 支持编辑器配色方案
    - 基于Kate的语法显示支持
    - 基于全文的单词自动完成
    - 支持键盘快捷键绑定方案
    - Markdown文档编辑支持
        - 实时预览和同步显示
        - 自定义CSS显示
        - 可导出HTML和PDF文档

    - 批量转换/合并为HTML/PDF文档

LiteIDE安装配置

- LiteIDE安装
    - 下载地址 http://sourceforge.net/projects/liteide/files/>
    - 源码地址 https://github.com/visualfc/liteide

    首先安装好Go语言环境，然后根据操作系统下载LiteIDE对应的压缩文件直接解压即可使用。

- 编译环境设置

  根据自身系统要求切换和配置LiteIDE当前使用的环境变量。

  以Windows操作系统，64位Go语言为例， 工具栏的环境配置中选择win64，点 `编辑环境` ， 进入LiteIDE编辑win64.env文件

```
GOROOT=c:\go
GOBIN=
GOARCH=amd64
GOOS=windows
CGO_ENABLED=1

PATH=%GOBIN%;%GOROOT%\bin;%PATH%
。。。
```

  将其中的 `GOROOT=c:\go` 修改为当前Go安装路径，存盘即可，如果有MinGW64，可以将 `c:\MinGW64\bin` 加入PATH中以便go调用gcc支持CGO编译。

  以Linux操作系统，64位Go语言为例， 工具栏的环境配置中选择linux64，点 `编辑环境` ， 进入LiteIDE编辑linux64.env文件

```
GOROOT=$HOME/go
GOBIN=
GOARCH=amd64
GOOS=linux
CGO_ENABLED=1

PATH=$GOBIN:$GOROOT/bin:$PATH
。。。
```

  将其中的 `GOROOT=$HOME/go` 修改为当前Go安装路径，存盘即可。

- GOPATH设置

Go语言的工具链使用GOPATH设置，是Go语言开发的项目路径列表，在命令行中输入(在LiteIDE中也可以 `Ctrl+,` 直接输入) `go help gopath` 快速查看GOPATH文档。

在LiteIDE中可以方便的查看和设置GOPATH。通过 `菜单－查看－GOPATH` 设置，可以查看系统中已存在的GOPATH列表， 同时可根据需要添加项目目录到自定义GOPATH列表中。

Sublime Text

这里将介绍Sublime Text 2（以下简称Sublime）+GoSublime的组合，那么为什么选择这个组合呢？

- 自动化提示代码,如下图所示

  图1.5 sublime自动化提示界面

- 保存的时候自动格式化代码，让您编写的代码更加美观，符合Go的标准。
- 支持项目管理

  图1.6 sublime项目管理界面

- 支持语法高亮
- Sublime Text 2可免费使用，只是保存次数达到一定数量之后就会提示是否购买，点击取消继续用，和正式注册版本没有任何区别。

接下来就开始讲如何安装，下载Sublime

根据自己相应的系统下载相应的版本，然后打开Sublime，对于不熟悉Sublime的同学可以先看一下这篇文章Sublime Text 2 入门及技巧

1. 打开之后安装 Package Control：Ctrl+` 打开命令行，执行如下代码：

   import urllib2,os; pf='Package Control.sublime-package'; ipp=sublime.installed_packages_path(); os.makedirs(ipp) if not os.path.exists(ipp) else None; urllib2.install_opener(urllib2.build_opener(urllib2.ProxyHandler())); open(os.path.join(ipp,pf),'wb').write(urllib2.urlopen('http://sublime.wbond.net/'+pf.replace(' ','%20')).read()); print 'Please restart Sublime Text to finish installation'

   这个时候重启一下Sublime，可以发现在在菜单栏多了一个如下的栏目，说明Package Control已经安装成功了。

图1.7 sublime包管理

1. 安装完之后就可以安装Sublime的插件了。需安装GoSublime、SidebarEnhancements和Go Build，安装插件之后记得重启Sublime生效，Ctrl+Shift+p打开Package Controll 输入 `pcip` （即“Package Control: Install Package”的缩写）。

   这个时候看左下角显示正在读取包数据，完成之后出现如下界面

   图1.8 sublime安装插件界面

   这个时候输入GoSublime，按确定就开始安装了。同理应用于SidebarEnhancements和Go Build。

2. 验证是否安装成功，你可以打开Sublime，打开main.go，看看语法是不是高亮了，输入 `import` 是不是自动化提示了， `import "fmt"` 之后，输入 `fmt.` 是不是自动化提示有函数了。

   如果已经出现这个提示，那说明你已经安装完成了，并且完成了自动提示。

   如果没有出现这样的提示，一般就是你的 `$PATH` 没有配置正确。你可以打开终端，输入gocode，是不是能够正确运行，如果不行就说明 `$PATH` 没有配置正确。 (针对XP)有时候在终端能运行成功,但sublime无提示或者编译解码错误,请安装sublime text3和convert utf8插件试一试

3. MacOS下已经设置了$GOROOT, $GOPATH, $GOBIN，还是没有自动提示怎么办。

   请在sublime中使用command + 9， 然后输入env检查$PATH, GOROOT, $GOPATH, $GOBIN等变量， 如果没有请采用下面的方法。

   首先建立下面的连接， 然后从Terminal中直接启动sublime

   ln -s /Applications/Sublime\ Text\ 2.app/Contents/SharedSupport/bin/subl /usr/local/bin/sublime

Vim

Vim是从vi发展出来的一个文本编辑器, 代码补全、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。

图1.9 VIM编辑器自动化提示Go界面

1. 配置vim高亮显示

```
cp -r $GOROOT/misc/vim/* ~/.vim/
```

2. 在~/.vimrc文件中增加语法高亮显示

```
filetype plugin indent on
syntax on
```

3. 安装Gocode

```
go get -u github.com/nsf/gocode
```

gocode默认安装到 `$GOPATH/bin` 下面。

4. 配置Gocode

```
~ cd $GOPATH/src/github.com/nsf/gocode/vim
~ ./update.bash
~ gocode set propose-builtins true
propose-builtins true
~ gocode set lib-path "/home/border/gocode/pkg/linux_amd64"
lib-path "/home/border/gocode/pkg/linux_amd64"
~ gocode set
propose-builtins true
lib-path "/home/border/gocode/pkg/linux_amd64"
```

> gocode set里面的两个参数的含意说明：
>
> propose-builtins：是否自动提示Go的内置函数、类型和常量，默认为false，不提示。
>
> lib-path:默认情况下，gocode只会搜索**$GOPATH/pkg/$GOOS_$GOARCH** 和 **$GOROOT/pkg/$GOOS_$GOARCH**目录下的包，当然这个设置就是可以设置我们额外的lib能访问的路径

1. 恭喜你，安装完成，你现在可以使用 `:e main.go` 体验一下开发Go的乐趣。

更多VIM 设定, 可参考链接

Emacs

Emacs传说中的神器，她不仅仅是一个编辑器，它是一个整合环境，或可称它为集成开发环境，这些功能如让使用者置身于全功能的操作系统中。

图1.10 Emacs编辑Go主界面

1. 配置Emacs高亮显示

```
cp $GOROOT/misc/emacs/* ~/.emacs.d/
```

2. 安装Gocode

```
go get -u github.com/nsf/gocode
```

gocode默认安装到 `$GOBIN` 里面下面。

3. 配置Gocode

```
~ cd $GOPATH/src/github.com/nsf/gocode/emacs
~ cp go-autocomplete.el ~/.emacs.d/
~ gocode set propose-builtins true
propose-builtins true
~ gocode set lib-path "/home/border/gocode/pkg/linux_amd64" //
lib-path "/home/border/gocode/pkg/linux_amd64"
~ gocode set
propose-builtins true
lib-path "/home/border/gocode/pkg/linux_amd64"
```

1. 需要安装 Auto Completion

下载AutoComplete并解压

~ make install DIR=$HOME/.emacs.d/auto-complete

配置~/.emacs文件

```
;;auto-complete
(require 'auto-complete-config)
(add-to-list 'ac-dictionary-directories "~/.emacs.d/auto-compl
(ac-config-default)
(local-set-key (kbd "M-/") 'semantic-complete-analyze-inline)
(local-set-key "." 'semantic-complete-self-insert)
(local-set-key ">" 'semantic-complete-self-insert)
```

详细信息参考: http://www.emacswiki.org/emacs/AutoComplete

2. 配置.emacs

```
;; golang mode
(require 'go-mode-load)
(require 'go-autocomplete)
;; speedbar
;; (speedbar 1)
```

```
(speedbar-add-supported-extension ".go")
(add-hook
'go-mode-hook
'(lambda ()
    ;; gocode
    (auto-complete-mode 1)
    (setq ac-sources '(ac-source-go))
    ;; Imenu & Speedbar
    (setq imenu-generic-expression
        '(("type" "^type *\\([^ \t\n\r\f]*\\)" 1)
        ("func" "^func *\\(.*\\) {" 1)))
    (imenu-add-to-menubar "Index")
    ;; Outline mode
    (make-local-variable 'outline-regexp)
    (setq outline-regexp "//\\.\\|//[^\r\n\f][^\r\n\f]\\|pack\
    (outline-minor-mode 1)
    (local-set-key "\M-a" 'outline-previous-visible-heading)
    (local-set-key "\M-e" 'outline-next-visible-heading)
    ;; Menu bar
    (require 'easymenu)
    (defconst go-hooked-menu
        '("Go tools"
        ["Go run buffer" go t]
        ["Go reformat buffer" go-fmt-buffer t]
        ["Go check buffer" go-fix-buffer t]))
    (easy-menu-define
        go-added-menu
        (current-local-map)
        "Go tools"
        go-hooked-menu)

    ;; Other
    (setq show-trailing-whitespace t)
    ))
;; helper function
(defun go ()
    "run current buffer"
    (interactive)
    (compile (concat "go run " (buffer-file-name))))

;; helper function
(defun go-fmt-buffer ()
    "run gofmt on current buffer"
    (interactive)
    (if buffer-read-only
    (progn
        (ding)
        (message "Buffer is read only"))
    (let ((p (line-number-at-pos))
    (filename (buffer-file-name))
    (old-max-mini-window-height max-mini-window-height))
        (show-all)
        (if (get-buffer "*Go Reformat Errors*")
```

```
        (progn
            (delete-windows-on "*Go Reformat Errors*")
            (kill-buffer "*Go Reformat Errors*")))
            (setq max-mini-window-height 1)
            (if (= 0 (shell-command-on-region (point-min) (point-m
        (progn
            (erase-buffer)
            (insert-buffer-substring "*Go Reformat Output*")
            (goto-char (point-min))
            (forward-line (1- p)))
        (with-current-buffer "*Go Reformat Errors*"
        (progn
            (goto-char (point-min))
            (while (re-search-forward "<standard input>" nil t)
            (replace-match filename))
            (goto-char (point-min))
            (compilation-mode))))
            (setq max-mini-window-height old-max-mini-window-heigh
            (delete-windows-on "*Go Reformat Output*")
            (kill-buffer "*Go Reformat Output*"))))
    ;; helper function
    (defun go-fix-buffer ()
        "run gofix on current buffer"
        (interactive)
        (show-all)
        (shell-command-on-region (point-min) (point-max) "go tool
```

3. 恭喜你，你现在可以体验在神器中开发Go的乐趣。默认speedbar是关闭的，如果打开需要把 ;; (speedbar 1) 前面的注释去掉，或者也可以通过 *M-x speedbar* 手动开启。

Eclipse

Eclipse也是非常常用的开发利器，以下介绍如何使用Eclipse来编写Go程序。

图1.11 Eclipse编辑Go的主界面

1. 首先下载并安装好Eclipse
2. 下载goclipse插件

   http://code.google.com/p/goclipse/wiki/InstallationInstructions

3. 下载gocode，用于go的代码补全提示

   gocode的github地址：

```
https://github.com/nsf/gocode
```

在windows下要安装git，通常用msysgit

再在cmd下安装：

```
go get -u github.com/nsf/gocode
```

也可以下载代码，直接用go build来编译，会生成gocode.exe

4. 下载MinGW并按要求装好
5. 配置插件

   Windows->Reference->Go

   (1).配置Go的编译器

   图1.12 设置Go的一些基础信息

(2).配置Gocode（可选，代码补全），设置Gocode路径为之前生成的gocode.exe文件

图1.13 设置gocode信息

(3).配置GDB（可选，做调试用），设置GDB路径为MingW安装目录下的gdb.exe文件

图1.14 设置GDB信息

1. 测试是否成功

   新建一个go工程，再建立一个hello.go。如下图：

   图1.15 新建项目编辑文件

   调试如下（要在console中用输入命令来调试）：

   图1.16 调试Go程序

IntelliJ IDEA

熟悉Java的读者应该对于idea不陌生，idea是通过一个插件来支持go语言的高亮语法,代码提示和重构实现。

1. 先下载idea，idea支持多平台：win,mac,linux，如果有钱就买个正式版，如果不行就使用社区免费版，对于只是开发Go语言来说免费版足够用了

2. 安装Go插件，点击菜单File中的Setting，找到Plugins,点击,Broswer repo按钮。国内的用户可能会报错，自己解决哈。

3. 这时候会看见很多插件，搜索找到Golang,双击,download and install。等到golang那一行后面出现Downloaded标志后,点OK。

   然后点 Apply .这时候IDE会要求你重启。

4. 重启完毕后,创建新项目会发现已经可以创建golang项目了：

   下一步,会要求你输入 go sdk的位置,一般都安装在C:\Go，linux和mac根据自己的安装目录设置，选中目录确定,就可以了。

links

- 目录
- 上一节: Go 命令
- 下一节: 总结

1.5 总结

这一章中我们主要介绍了如何安装Go，Go可以通过三种方式安装：源码安装、标准包安装、第三方工具安装，安装之后我们需要配置我们的开发环境，然后介绍了如何配置本地的 `$GOPATH` ，通过设置 `$GOPATH` 之后读者就可以创建项目，接着介绍了如何来进行项目编译、应用安装等问题，这些需要用到很多Go命令，所以接着就介绍了一些Go的常用命令工具，包括编译、安装、格式化、测试等命令，最后介绍了Go的开发工具，目前有很多Go的开发工具：LiteIDE、sublime、VIM、Emacs、Eclipse、Idea等工具，读者可以根据自己熟悉的工具进行配置，希望能够通过方便的工具快速的开发Go应用。

links

- 目录
- 上一节: Go开发工具
- 下一章: Go语言基础

2 Go语言基础

Go是一门类似C的编译型语言，但是它的编译速度非常快。这门语言的关键字总共也就二十五个，比英文字母还少一个，这对于我们的学习来说就简单了很多。先让我们看一眼这些关键字都长什么样：

```
break    default     func   interface select
case     defer       go     map       struct
chan     else        goto   package   switch
const    fallthrough if     range     type
continue for         import return    var
```

在接下来的这一章中，我将带领你去学习这门语言的基础。通过每一小节的介绍，你将发现，Go的世界是那么地简洁，设计是如此地美妙，编写Go将会是一件愉快的事情。等回过头来，你就会发现这二十五个关键字是多么地亲切。

目录

links

- 目录
- 上一章: 第一章总结
- 下一节: 你好，Go

2.1 你好，Go

在开始编写应用之前，我们先从最基本的程序开始。就像你造房子之前不知道什么是地基一样，编写程序也不知道如何开始。因此，在本节中，我们要学习用最基本的语法让Go程序运行起来。

程序

这就像一个传统，在学习大部分语言之前，你先学会如何编写一个可以输出 `hello world` 的程序。

准备好了吗？Let's Go!

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, world or 你好，世界 or καλημ´ρα κόσμ or こんに
}
```

输出如下：

```
Hello, world or 你好，世界 or καλημ´ρα κόσμ or こんにちはせかい
```

详解

首先我们要了解一个概念，Go程序是通过 `package` 来组织的

`package <pkgName>` （在我们的例子中是 `package main` ）这一行告诉我们当前文件属于哪个包，而包名 `main` 则告诉我们它是一个可独立运行的包，它在编译后会产生可执行文件。除了 `main` 包之外，其它的包最后都会生成 `*.a` 文件（也就是包文件）并放置在 `$GOPATH/pkg/$GOOS_$GOARCH` 中（以Mac为例就是 `$GOPATH/pkg/darwin_amd64` ）。

> 每一个可独立运行的Go程序，必定包含一个 `package main` ，在这个 `main` 包中必定包含一个入口函数 `main` ，而这个函数既没有参数，也没有返回值。

为了打印 `Hello, world...` ，我们调用了一个函数 `Printf` ，这个函数来自于 `fmt` 包，所以我们在第三行中导入了系统级别的 `fmt` 包： `import "fmt"` 。

包的概念和Python中的package类似，它们都有一些特别的好处：模块化（能够把你的程序分成多个模块)和可重用性（每个模块都能被其它应用程序反复使用）。我们在这里只是先了解一下包的概念，后面我们将会编写自己的包。

在第五行中，我们通过关键字 `func` 定义了一个 `main` 函数，函数体被放在 `{}` （大括号）中，就像我们平时写C、C++或Java时一样。

大家可以看到 `main` 函数是没有任何的参数的，我们接下来就学习如何编写带参数的、返回0个或多个值的函数。

第六行，我们调用了 `fmt` 包里面定义的函数 `Printf` 。大家可以看到，这个函数是通过 `<pkgName>.<funcName>` 的方式调用的，这一点和Python十分相似。

> 前面提到过，包名和包所在的文件夹名可以是不同的，此处的 `<pkgName>` 即为通过 `package <pkgName>` 声明的包名，而非文件夹名。

最后大家可以看到我们输出的内容里面包含了很多非ASCII码字符。实际上，Go是天生支持UTF-8的，任何字符都可以直接输出，你甚至可以用UTF-8中的任何字符作为标识符。

结论

Go使用 `package` （和Python的模块类似）来组织代码。 `main.main()` 函数(这个函数位于主包）是每一个独立的可运行程序的入口点。Go使用UTF-8字符串和标识符(因为UTF-8的发明者也就是Go的发明者)，所以它天生支持多语言。

links

- 目录
- 上一节: Go语言基础
- 下一节: Go基础

2.2 Go基础

这小节我们将要介绍如何定义变量、常量、Go内置类型以及Go程序设计中的一些技巧。

定义变量

Go语言里面定义变量有多种方式。

使用 `var` 关键字是Go最基本的定义变量方式，与C语言不同的是Go把变量类型放在变量名后面：

```
//定义一个名称为“variableName”，类型为"type"的变量
var variableName type
```

定义多个变量

```
//定义三个类型都是“type”的变量
var vname1, vname2, vname3 type
```

定义变量并初始化值

```
//初始化“variableName”的变量为“value”值，类型是“type”
var variableName type = value
```

同时初始化多个变量

```
/*
    定义三个类型都是"type"的变量,并且分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
*/
var vname1, vname2, vname3 type= v1, v2, v3
```

你是不是觉得上面这样的定义有点繁琐？没关系，因为Go语言的设计者也发现了，有一种写法可以让它变得简单一点。我们可以直接忽略类型声明，那么上面的代码变成这样了：

```
/*
    定义三个变量，它们分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
    然后Go会根据其相应值的类型来帮你初始化它们
*/
var vname1, vname2, vname3 = v1, v2, v3
```

你觉得上面的还是有些繁琐？好吧，我也觉得。让我们继续简化：

```
/*
    定义三个变量，它们分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
    编译器会根据初始化的值自动推导出相应的类型
*/
vname1, vname2, vname3 := v1, v2, v3
```

现在是不是看上去非常简洁了？ `:=` 这个符号直接取代了 `var` 和 `type` ,这种形式叫做简短声明。不过它有一个限制，那就是它只能用在函数内部；在函数外部使用则会无法编译通过，所以一般用 `var` 方式来定义全局变量。

`_` （下划线）是个特殊的变量名，任何赋予它的值都会被丢弃。在这个例子中，我们将值 `35` 赋予 `b` ，并同时丢弃 `34` ：

```
_, b := 34, 35
```

Go对于已声明但未使用的变量会在编译阶段报错，比如下面的代码就会产生一个错误：声明了 `i` 但未使用。

```
package main

func main() {
    var i int
}
```

常量

所谓常量，也就是在程序编译阶段就确定下来的值，而程序在运行时无法改变该值。在Go程序中，常量可定义为数值、布尔值或字符串等类型。

它的语法如下：

```
const constantName = value
//如果需要，也可以明确指定常量的类型：
const Pi float32 = 3.1415926
```

下面是一些常量声明的例子：

```
const Pi = 3.1415926
const i = 10000
const MaxThread = 10
const prefix = "astaxie_"
```

Go 常量和一般程序语言不同的是，可以指定相当多的小数位数(例如200位)， 若指定給float32自动缩短为32bit，指定给float64自动缩短为64bit，详情参考链接

内置基础类型

Boolean

在Go中，布尔值的类型为 `bool` ，值是 `true` 或 `false` ，默认为 `false` 。

```
//示例代码
var isActive bool  // 全局变量声明
var enabled, disabled = true, false  // 忽略类型的声明
func test() {
    var available bool  // 一般声明
    valid := false      // 简短声明
    available = true    // 赋值操作
}
```

数值类型

整数类型有无符号和带符号两种。Go同时支持 `int` 和 `uint` ，这两种类型的长度相同，但具体长度取决于不同编译器的实现。Go里面也有直接定义好位数的类型： `rune` ， `int8` ， `int16` ， `int32` ， `int64` 和 `byte` ， `uint8` ， `uint16` ， `uint32` ， `uint64` 。其中 `rune` 是 `int32` 的别称， `byte` 是 `uint8` 的别称。

> 需要注意的一点是，这些类型的变量之间不允许互相赋值或操作，不然会在编译时引起编译器报错。
>
> 如下的代码会产生错误：invalid operation: a + b (mismatched types int8 and int32)
>
> > var a int8
> >
> > var b int32
> >
> > c:=a + b
>
> 另外，尽管int的长度是32 bit, 但int 与 int32并不可以互用。

浮点数的类型有`float32`和`float64`两种（没有`float`类型），默认是`float64`。

这就是全部吗？No！Go还支持复数。它的默认类型是`complex128`（64位实数+64位虚数）。如果需要小一些的，也有`complex64` (32位实数+32位虚数)。复数的形式为`RE + IMi`，其中`RE`是实数部分，`IM`是虚数部分，而最后的`i`是虚数单位。下面是一个使用复数的例子：

```
var c complex64 = 5+5i
//output: (5+5i)
fmt.Printf("Value is: %v", c)
```

字符串

我们在上一节中讲过，Go中的字符串都是采用`UTF-8`字符集编码。字符串是用一对双引号（`""`）或反引号（`` ` `` `` ` ``）括起来定义，它的类型是`string`。

```
//示例代码
var frenchHello string  // 声明变量为字符串的一般方法
var emptyString string = ""  // 声明了一个字符串变量，初始化为空字符串
func test() {
    no, yes, maybe := "no", "yes", "maybe"  // 简短声明，同时声明多个变
    japaneseHello := "Konichiwa"  // 同上
    frenchHello = "Bonjour"  // 常规赋值
}
```

在Go中字符串是不可变的，例如下面的代码编译时会报错：cannot assign to s[0]

```
var s string = "hello"
s[0] = 'c'
```

但如果真的想要修改怎么办呢？下面的代码可以实现：

```
s := "hello"
c := []byte(s)  // 将字符串 s 转换为 []byte 类型
c[0] = 'c'
s2 := string(c)  // 再转换回 string 类型
fmt.Printf("%s\n", s2)
```

Go中可以使用 `+` 操作符来连接两个字符串：

```
s := "hello,"
m := " world"
a := s + m
fmt.Printf("%s\n", a)
```

修改字符串也可写为：

```
s := "hello"
s = "c" + s[1:] // 字符串虽不能更改，但可进行切片操作
fmt.Printf("%s\n", s)
```

如果要声明一个多行的字符串怎么办？可以通过 `` ` `` 来声明：

```
m := `hello
    world`
```

`` ` `` 括起的字符串为Raw字符串，即字符串在代码中的形式就是打印时的形式，它没有字符转义，换行也将原样输出。例如本例中会输出：

```
hello
    world
```

错误类型

Go内置有一个 `error` 类型，专门用来处理错误信息，Go的 `package` 里面还专门有一个包 `errors` 来处理错误：

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
    fmt.Print(err)
}
```

Go数据底层的存储

下面这张图来源于Russ Cox Blog中一篇介绍Go数据结构的文章，大家可以看到这些基础类型底层都是分配了一块内存，然后存储了相应的值。

图2.1 Go数据格式的存储

一些技巧

分组声明

在Go语言中，同时声明多个常量、变量，或者导入多个包时，可采用分组的方式进行声明。

例如下面的代码：

```
import "fmt"
import "os"

const i = 100
const pi = 3.1415
const prefix = "Go_"

var i int
var pi float32
var prefix string
```

可以分组写成如下形式：

```
import(
    "fmt"
    "os"
)

const(
    i = 100
    pi = 3.1415
    prefix = "Go_"
)

var(
    i int
    pi float32
    prefix string
)
```

iota枚举

Go里面有一个关键字 `iota` ，这个关键字用来声明 `enum` 的时候采用，它默认开始值是0，const中每增加一行加1：

```
const(
    x = iota  // x == 0
    y = iota  // y == 1
    z = iota  // z == 2
    w  // 常量声明省略值时，默认和之前一个值的字面相同。这里隐式地说w = iota
)

const v = iota // 每遇到一个const关键字，iota就会重置，此时v == 0

const (
  e, f, g = iota, iota, iota //e=0,f=0,g=0 iota在同一行值相同
)

const  (
    a = iota    a=0
    b = "B"
    c = iota    //c=2
    d,e,f = iota,iota,iota //d=3,e=3,f=3
    g //g = 4
)
```

> 除非被显式设置为其它值或 `iota` ，每个 `const` 分组的第一个常量被默认设置为它的0值，第二及后续的常量被默认设置为它前面那个常量的值，如果前面那个常量的值是 `iota` ，则它也被设置为 `iota` 。

Go程序设计的一些规则

Go之所以会那么简洁，是因为它有一些默认的行为：

- 大写字母开头的变量是可导出的，也就是其它包可以读取的，是公用变量；小写字母开头的就是不可导出的，是私有变量。
- 大写字母开头的函数也是一样，相当于 `class` 中的带 `public` 关键词的公有函数；小写字母开头的就是有 `private` 关键词的私有函数。

array、slice、map

array

`array` 就是数组，它的定义方式如下：

```
var arr [n]type
```

在 `[n]type` 中， `n` 表示数组的长度， `type` 表示存储元素的类型。对数组的操作和其它语言类似，都是通过 `[]` 来进行读取或赋值：

```
var arr [10]int  // 声明了一个int类型的数组
arr[0] = 42      // 数组下标是从0开始的
arr[1] = 13      // 赋值操作
fmt.Printf("The first element is %d\n", arr[0])  // 获取数据，返回42
fmt.Printf("The last element is %d\n", arr[9]) //返回未赋值的最后一个元
```

由于长度也是数组类型的一部分，因此 `[3]int` 与 `[4]int` 是不同的类型，数组也就不能改变长度。数组之间的赋值是值的赋值，即当把一个数组作为参数传入函数的时候，传入的其实是该数组的副本，而不是它的指针。如果要使用指针，那么就需要用到后面介绍的 `slice` 类型了。

数组可以使用另一种 `:=` 来声明

```
a := [3]int{1, 2, 3} // 声明了一个长度为3的int数组

b := [10]int{1, 2, 3} // 声明了一个长度为10的int数组，其中前三个元素初始化

c := [...]int{4, 5, 6} // 可以省略长度而采用`...`的方式，Go会自动根据元素
```

也许你会说，我想数组里面的值还是数组，能实现吗？当然咯，Go支持嵌套数组，即多维数组。比如下面的代码就声明了一个二维数组：

```
// 声明了一个二维数组，该数组以两个数组作为元素，其中每个数组中又有4个int类型的
doubleArray := [2][4]int{[4]int{1, 2, 3, 4}, [4]int{5, 6, 7, 8}}

// 上面的声明可以简化，直接忽略内部的类型
easyArray := [2][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}}
```

数组的分配如下所示：



图2.2 多维数组的映射关系

slice

在很多应用场景中，数组并不能满足我们的需求。在初始定义数组时，我们并不知道需要多大的数组，因此我们就需要“动态数组”。在Go里面这种数据结构叫 `slice`

`slice` 并不是真正意义上的动态数组，而是一个引用类型。 `slice` 总是指向一个底层 `array` ， `slice` 的声明也可以像 `array` 一样，只是不需要长度。

```
// 和声明array一样，只是少了长度
var fslice []int
```

接下来我们可以声明一个 `slice` ，并初始化数据，如下所示：

```
slice := []byte {'a', 'b', 'c', 'd'}
```

`slice` 可以从一个数组或一个已经存在的 `slice` 中再次声明。 `slice` 通过 `array[i:j]` 来获取，其中 `i` 是数组的开始位置， `j` 是结束位置，但不包含 `array[j]` ，它的长度是 `j-i` 。

```
// 声明一个含有10个元素元素类型为byte的数组
var ar = [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j

// 声明两个含有byte的slice
var a, b []byte

// a指向数组的第3个元素开始，并到第五个元素结束，
a = ar[2:5]
//现在a含有的元素: ar[2]、ar[3]和ar[4]

// b是数组ar的另一个slice
b = ar[3:5]
// b的元素是：ar[3]和ar[4]
```

> 注意 `slice` 和数组在声明时的区别：声明数组时，方括号内写明了数组的长度或使用 `...` 自动计算长度，而声明 `slice` 时，方括号内没有任何字符。

它们的数据结构如下所示

图2.3 slice和array的对应关系图

slice有一些简便的操作

- `slice` 的默认开始位置是0， `ar[:n]` 等价于 `ar[0:n]`
- `slice` 的第二个序列默认是数组的长度， `ar[n:]` 等价于 `ar[n:len(ar)]`
- 如果从一个数组里面直接获取 `slice` ，可以这样 `ar[:]` ，因为默认第一个序列是0，第二个是数组的长度，即等价于 `ar[0:len(ar)]`

下面这个例子展示了更多关于 `slice` 的操作：

```
// 声明一个数组
var array = [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
// 声明两个slice
var aSlice, bSlice []byte

// 演示一些简便操作
aSlice = array[:3] // 等价于aSlice = array[0:3] aSlice包含元素: a,b,c
aSlice = array[5:] // 等价于aSlice = array[5:10] aSlice包含元素: f,g,
aSlice = array[:]  // 等价于aSlice = array[0:10] 这样aSlice包含了全部的

// 从slice中获取slice
aSlice = array[3:7]  // aSlice包含元素: d,e,f,g, len=4, cap=7
bSlice = aSlice[1:3] // bSlice 包含aSlice[1], aSlice[2] 也就是含有: e
bSlice = aSlice[:3]  // bSlice 包含 aSlice[0], aSlice[1], aSlice[2]
bSlice = aSlice[0:5] // 对slice的slice可以在cap范围内扩展, 此时bSlice包
bSlice = aSlice[:]   // bSlice包含所有aSlice的元素: d,e,f,g
```

`slice` 是引用类型，所以当引用改变其中元素的值时，其它的所有引用都会改变该值，例如上面的 `aSlice` 和 `bSlice` ，如果修改了 `aSlice` 中元素的值，那么 `bSlice` 相对应的值也会改变。

从概念上面来说 `slice` 像一个结构体，这个结构体包含了三个元素：

- 一个指针，指向数组中 `slice` 指定的开始位置
- 长度，即 `slice` 的长度
- 最大长度，也就是 `slice` 开始位置到数组的最后位置的长度

  ```
  Array_a := [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', '
  Slice_a := Array_a[2:5]
  ```

上面代码的真正存储结构如下图所示

图2.4 slice对应数组的信息

对于 `slice` 有几个有用的内置函数：

- `len` 获取 `slice` 的长度
- `cap` 获取 `slice` 的最大容量
- `append` 向 `slice` 里面追加一个或者多个元素，然后返回一个和 `slice` 一样类型的 `slice`
- `copy` 函数 `copy` 从源 `slice` 的 `src` 中复制元素到目标 `dst` ，并且返回复制的元素的个数

注： `append` 函数会改变 `slice` 所引用的数组的内容，从而影响到引用同一数组的其它 `slice` 。 但当 `slice` 中没有剩余空间（即 `(cap-len) == 0` ）时，此时将动态分配新的数组空间。返回的 `slice` 数组指针将指向这个空间，而原数组的内容将保持不变；其它引用此数组的 `slice` 则不受影响。

从Go1.2开始slice支持了三个参数的slice，之前我们一直采用这种方式在slice或者array基础上来获取一个slice

```
var array [10]int
slice := array[2:4]
```

这个例子里面slice的容量是8，新版本里面可以指定这个容量

```
slice = array[2:4:7]
```

上面这个的容量就是 `7-2` ，即5。这样这个产生的新的slice就没办法访问最后的三个元素。

如果slice是这样的形式 `array[:i:j]` ，即第一个参数为空，默认值就是0。

map

`map` 也就是Python中字典的概念，它的格式为 `map[keyType]valueType`

我们看下面的代码， `map` 的读取和设置也类似 `slice` 一样，通过 `key` 来操作，只是 `slice` 的 `index` 只能是｀int｀类型，而 `map` 多了很多类型，可以是 `int` ，可以是 `string` 及所有完全定义了 `==` 与 `!=` 操作的类型。

```
// 声明一个key是字符串，值为int的字典,这种方式的声明需要在使用之前使用make初始
var numbers map[string]int
// 另一种map的声明方式
numbers := make(map[string]int)
numbers["one"] = 1  //赋值
numbers["ten"] = 10 //赋值
numbers["three"] = 3

fmt.Println("第三个数字是: ", numbers["three"]) // 读取数据
// 打印出来如:第三个数字是: 3
```

这个 `map` 就像我们平常看到的表格一样，左边列是 `key` ，右边列是值

使用map过程中需要注意的几点：

- `map` 是无序的，每次打印出来的 `map` 都会不一样，它不能通过 `index` 获取，而必须通过 `key` 获取

- `map` 的长度是不固定的，也就是和 `slice` 一样，也是一种引用类型
- 内置的 `len` 函数同样适用于 `map` ，返回 `map` 拥有的 `key` 的数量
- `map` 的值可以很方便的修改，通过 `numbers["one"]=11` 可以很容易的把key为 `one` 的字典值改为 `11`
- `map` 和其他基本型别不同，它不是thread-safe，在多个go-routine存取时，必须使用mutex lock机制

`map` 的初始化可以通过 `key:val` 的方式初始化值，同时 `map` 内置有判断是否存在 `key` 的方式

通过 `delete` 删除 `map` 的元素：

```
// 初始化一个字典
rating := map[string]float32{"C":5, "Go":4.5, "Python":4.5, "C++":2
// map有两个返回值，第二个返回值，如果不存在key，那么ok为false，如果存在ok为
csharpRating, ok := rating["C#"]
if ok {
    fmt.Println("C# is in the map and its rating is ", csharpRating
} else {
    fmt.Println("We have no rating associated with C# in the map")
}

delete(rating, "C")  // 删除key为C的元素
```

上面说过了， `map` 也是一种引用类型，如果两个 `map` 同时指向一个底层，那么一个改变，另一个也相应的改变：

```
m := make(map[string]string)
m["Hello"] = "Bonjour"
m1 := m
m1["Hello"] = "Salut"  // 现在m["hello"]的值已经是Salut了
```

make、new操作

`make` 用于内建类型（ `map` 、 `slice` 和 `channel` ）的内存分配。 `new` 用于各种类型的内存分配。

内建函数 `new` 本质上说跟其它语言中的同名函数功能一样： `new(T)` 分配了零值填充的 `T` 类型的内存空间，并且返回其地址，即一个 `*T` 类型的值。用Go的术语说，它返回了一个指针，指向新分配的类型 `T` 的零值。有一点非常重要：

> `new` 返回指针。

内建函数 `make(T, args)` 与 `new(T)` 有着不同的功能，make只能创建 `slice` 、 `map` 和 `channel` ，并且返回一个有初始值(非零)的 `T` 类型，而不是 `*T` 。本质来讲，导致这三个类型有所不同的原因是指向数据结构的引用在使用

前必须被初始化。例如，一个 `slice` ，是一个包含指向数据（内部 `array` ）的指针、长度和容量的三项描述符；在这些项目被初始化之前， `slice` 为 `nil` 。对于 `slice` 、 `map` 和 `channel` 来说， `make` 初始化了内部的数据结构，填充适当的值。

> `make` 返回初始化后的（非零）值。

下面这个图详细的解释了 `new` 和 `make` 之间的区别。

图2.5 make和new对应底层的内存分配

零值

关于“零值”，所指并非是空值，而是一种“变量未填充前”的默认值，通常为0。 此处罗列 部分类型 的 “零值”

```
int     0
int8    0
int32   0
int64   0
uint    0x0
rune    0 //rune的实际类型是 int32
byte    0x0 // byte的实际类型是 uint8
float32 0 //长度为 4 byte
float64 0 //长度为 8 byte
bool    false
string  ""
```

links

- 目录
- 上一章: 你好,Go
- 下一节: 流程和函数

2.3 流程和函数

这小节我们要介绍Go里面的流程控制以及函数操作。

流程控制

流程控制在编程语言中是最伟大的发明了，因为有了它，你可以通过很简单的流程描述来表达很复杂的逻辑。Go中流程控制分三大类：条件判断，循环控制和无条件跳转。

if

`if` 也许是各种编程语言中最常见的了，它的语法概括起来就是:如果满足条件就做某事，否则做另一件事。

Go里面 `if` 条件判断语句中不需要括号，如下代码所示

```
if x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}
```

Go的 `if` 还有一个强大的地方就是条件判断语句里面允许声明一个变量，这个变量的作用域只能在该条件逻辑块内，其他地方就不起作用了，如下所示

```
// 计算获取值x,然后根据x返回的大小，判断是否大于10。
if x := computedValue(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}

//这个地方如果这样调用就编译出错了，因为x是条件里面的变量
fmt.Println(x)
```

多个条件的时候如下所示：

```
if integer == 3 {
    fmt.Println("The integer is equal to 3")
} else if integer < 3 {
    fmt.Println("The integer is less than 3")
} else {
    fmt.Println("The integer is greater than 3")
}
```

goto

Go有 `goto` 语句——请明智地使用它。用 `goto` 跳转到必须在当前函数内定义的标签。例如假设这样一个循环：

```
func myFunc() {
    i := 0
Here:   //这行的第一个词，以冒号结束作为标签
    println(i)
    i++
    goto Here   //跳转到Here去
}
```

> 标签名是大小写敏感的。

for

Go里面最强大的一个控制逻辑就是 `for` ，它即可以用来循环读取数据，又可以当作 `while` 来控制逻辑，还能迭代操作。它的语法如下：

```
for expression1; expression2; expression3 {
    //...
}
```

`expression1` 、 `expression2` 和 `expression3` 都是表达式，其中 `expression1` 和 `expression3` 是变量声明或者函数调用返回值之类的， `expression2` 是用来条件判断， `expression1` 在循环开始之前调用， `expression3` 在每轮循环结束之时调用。

一个例子比上面讲那么多更有用，那么我们看看下面的例子吧：

```
package main
import "fmt"

func main(){
    sum := 0;
    for index:=0; index < 10 ; index++ {
        sum += index
    }
    fmt.Println("sum is equal to ", sum)
}
// 输出：sum is equal to 45
```

有些时候需要进行多个赋值操作，由于Go里面没有 `,` 操作符，那么可以使用平行赋值 `i, j = i+1, j-1`

有些时候如果我们忽略 `expression1` 和 `expression3` ：

```
sum := 1
for ; sum < 1000;  {
    sum += sum
}
```

其中 `;` 也可以省略，那么就变成如下的代码了，是不是似曾相识？对，这就是 `while` 的功能。

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

在循环里面有两个关键操作 `break` 和 `continue` ，`break` 操作是跳出当前循环， `continue` 是跳过本次循环。当嵌套过深的时候， `break` 可以配合标签使用，即跳转至标签所指定的位置，详细参考如下例子：

```
for index := 10; index>0; index-- {
    if index == 5{
        break // 或者continue
    }
    fmt.Println(index)
}
// break打印出来10、9、8、7、6
// continue打印出来10、9、8、7、6、4、3、2、1
```

`break` 和 `continue` 还可以跟着标号，用来跳到多重循环中的外层循环

`for` 配合 `range` 可以用于读取 `slice` 和 `map` 的数据：

```
for k,v:=range map {
    fmt.Println("map's key:",k)
    fmt.Println("map's val:",v)
}
```

由于 Go 支持 “多值返回”, 而对于“声明而未被调用”的变量, 编译器会报错, 在这种情况下, 可以使用 `_` 来丢弃不需要的返回值 例如

```
for _, v := range map{
    fmt.Println("map's val:", v)
}
```

switch

有些时候你需要写很多的 `if-else` 来实现一些逻辑处理，这个时候代码看上去就很丑很冗长，而且也不易于以后的维护，这个时候 `switch` 就能很好的解决这个问题。它的语法如下

```
switch sExpr {
case expr1:
    some instructions
case expr2:
    some other instructions
case expr3:
    some other instructions
default:
    other code
}
```

`sExpr` 和 `expr1` 、 `expr2` 、 `expr3` 的类型必须一致。Go的 `switch` 非常灵活，表达式不必是常量或整数，执行的过程从上至下，直到找到匹配项；而如果 `switch` 没有表达式，它会匹配 `true` 。

```
i := 10
switch i {
case 1:
    fmt.Println("i is equal to 1")
case 2, 3, 4:
    fmt.Println("i is equal to 2, 3 or 4")
case 10:
    fmt.Println("i is equal to 10")
default:
    fmt.Println("All I know is that i is an integer")
}
```

在第5行中，我们把很多值聚合在了一个 `case` 里面，同时，Go里面 `switch` 默认相当于每个 `case` 最后带有 `break` ，匹配成功后不会自动向下执行其他case，而是跳出整个 `switch` , 但是可以使用 `fallthrough` 强制执行后面的case代码。

```
integer := 6
switch integer {
case 4:
    fmt.Println("The integer was <= 4")
    fallthrough
case 5:
    fmt.Println("The integer was <= 5")
    fallthrough
case 6:
    fmt.Println("The integer was <= 6")
    fallthrough
case 7:
    fmt.Println("The integer was <= 7")
    fallthrough
case 8:
    fmt.Println("The integer was <= 8")
    fallthrough
default:
    fmt.Println("default case")
}
```

上面的程序将输出

```
The integer was <= 6
The integer was <= 7
The integer was <= 8
default case
```

函数

函数是Go里面的核心设计，它通过关键字 `func` 来声明，它的格式如下：

```
func funcName(input1 type1, input2 type2) (output1 type1, output2 t
    //这里是处理逻辑代码
    //返回多个值
    return value1, value2
}
```

上面的代码我们看出

- 关键字 `func` 用来声明一个函数 `funcName`
- 函数可以有一个或者多个参数，每个参数后面带有类型，通过 `,` 分隔
- 函数可以返回多个值
- 上面返回值声明了两个变量 `output1` 和 `output2` ，如果你不想声明也可以，直接就两个类型
- 如果只有一个返回值且不声明返回值变量，那么你可以省略 包括返回值 的括号
- 如果没有返回值，那么就直接省略最后的返回信息
- 如果有返回值， 那么必须在函数的外层添加return语句

下面我们来看一个实际应用函数的例子（用来计算Max值）

```
package main
import "fmt"

// 返回a、b中最大值.
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

func main() {
    x := 3
    y := 4
    z := 5

    max_xy := max(x, y) //调用函数max(x, y)
    max_xz := max(x, z) //调用函数max(x, z)

    fmt.Printf("max(%d, %d) = %d\n", x, y, max_xy)
    fmt.Printf("max(%d, %d) = %d\n", x, z, max_xz)
    fmt.Printf("max(%d, %d) = %d\n", y, z, max(y,z)) // 也可在这直接
}
```

上面这个里面我们可以看到 `max` 函数有两个参数，它们的类型都是 `int` ，那么第一个变量的类型可以省略（即 a,b int,而非 a int, b int)，默认为离它最近的类型，同理多于2个同类型的变量或者返回值。同时我们注意到它的返回值就是一个类型，这个就是省略写法。

多个返回值

Go语言比C更先进的特性，其中一点就是函数能够返回多个值。

我们直接上代码看例子

```
package main
import "fmt"

//返回 A+B 和 A*B
func SumAndProduct(A, B int) (int, int) {
    return A+B, A*B
}

func main() {
    x := 3
    y := 4

    xPLUSy, xTIMESy := SumAndProduct(x, y)

    fmt.Printf("%d + %d = %d\n", x, y, xPLUSy)
    fmt.Printf("%d * %d = %d\n", x, y, xTIMESy)
}
```

上面的例子我们可以看到直接返回了两个参数，当然我们也可以命名返回参数的变量，这个例子里面只是用了两个类型，我们也可以改成如下这样的定义，然后返回的时候不用带上变量名，因为直接在函数里面初始化了。但如果你的函数是导出的(首字母大写)，官方建议：最好命名返回值，因为不命名返回值，虽然使得代码更加简洁了，但是会造成生成的文档可读性差。

```
func SumAndProduct(A, B int) (add int, Multiplied int) {
    add = A+B
    Multiplied = A*B
    return
}
```

变参

Go函数支持变参。接受变参的函数是有着不定数量的参数的。为了做到这点，首先需要定义函数使其接受变参：

```
func myfunc(arg ...int) {}
```

`arg ...int` 告诉Go这个函数接受不定数量的参数。注意，这些参数的类型全部是 `int` 。在函数体中，变量 `arg` 是一个 `int` 的 `slice` ：

```
for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
}
```

传值与传指针

当我们传一个参数值到被调用函数里面时，实际上是传了这个值的一份copy，当在被调用函数中修改参数值的时候，调用函数中相应实参不会发生任何变化，因为数值变化只作用在copy上。

为了验证我们上面的说法，我们来看一个例子

```
package main
import "fmt"

//简单的一个函数，实现了参数+1的操作
func add1(a int) int {
    a = a+1 // 我们改变了a的值
    return a //返回一个新值
}

func main() {
    x := 3

    fmt.Println("x = ", x)  // 应该输出 "x = 3"

    x1 := add1(x)  //调用add1(x)

    fmt.Println("x+1 = ", x1) // 应该输出"x+1 = 4"
    fmt.Println("x = ", x)    // 应该输出"x = 3"
}
```

看到了吗？虽然我们调用了 `add1` 函数，并且在 `add1` 中执行 `a = a+1` 操作，但是上面例子中 `x` 变量的值没有发生变化

理由很简单：因为当我们调用 `add1` 的时候， `add1` 接收的参数其实是 `x` 的copy，而不是 `x` 本身。

那你也许会问了，如果真的需要传这个 `x` 本身,该怎么办呢？

这就牵扯到了所谓的指针。我们知道，变量在内存中是存放于一定地址上的，修改变量实际是修改变量地址处的内存。只有 `add1` 函数知道 `x` 变量所在的地址，才能修改 `x` 变量的值。所以我们需要将 `x` 所在地址 `&x` 传入函数，并将函数的参数的类型由 `int` 改为 `*int` ，即改为指针类型，才能在函数中修改 `x` 变量的值。此时参数仍然是按copy传递的，只是copy的是一个指针。请看下面的例子

```
package main
import "fmt"

//简单的一个函数，实现了参数+1的操作
func add1(a *int) int { // 请注意，
    *a = *a+1 // 修改了a的值
    return *a // 返回新值
}

func main() {
    x := 3

    fmt.Println("x = ", x)  // 应该输出 "x = 3"

    x1 := add1(&x)  // 调用 add1(&x) 传x的地址

    fmt.Println("x+1 = ", x1) // 应该输出 "x+1 = 4"
    fmt.Println("x = ", x)    // 应该输出 "x = 4"
}
```

这样，我们就达到了修改 `x` 的目的。那么到底传指针有什么好处呢？

- 传指针使得多个函数能操作同一个对象。
- 传指针比较轻量级 (8bytes),只是传内存地址，我们可以用指针传递体积大的结构体。如果用参数值传递的话, 在每次copy上面就会花费相对较多的系统开销（内存和时间）。所以当你要传递大的结构体的时候，用指针是一个明智的选择。
- Go语言中 `channel` ， `slice` ， `map` 这三种类型的实现机制类似指针，所以可以直接传递，而不用取地址后传递指针。（注：若函数需改变 `slice` 的长度，则仍需要取地址传递指针）

defer

Go语言中有种不错的设计，即延迟（defer）语句，你可以在函数中添加多个defer语句。当函数执行到最后时，这些defer语句会按照逆序执行，最后该函数返回。特别是当你在进行一些打开资源的操作时，遇到错误需要提前返回，在返回前你需要关闭相应的资源，不然很容易造成资源泄露等问题。如下代码所示，我们一般写打开一个资源是这样操作的：

```
func ReadWrite() bool {
    file.Open("file")
// 做一些工作
    if failureX {
        file.Close()
        return false
    }

    if failureY {
        file.Close()
        return false
    }

    file.Close()
    return true
}
```

我们看到上面有很多重复的代码，Go的 `defer` 有效解决了这个问题。使用它后，不但代码量减少了很多，而且程序变得更优雅。在 `defer` 后指定的函数会在函数退出前调用。

```
func ReadWrite() bool {
    file.Open("file")
    defer file.Close()
    if failureX {
        return false
    }
    if failureY {
        return false
    }
    return true
}
```

如果有很多调用 `defer` ，那么 `defer` 是采用后进先出模式，所以如下代码会输出 `4 3 2 1 0`

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

函数作为值、类型

在Go中函数也是一种变量，我们可以通过 `type` 来定义它，它的类型就是所有拥有相同的参数，相同的返回值的一种类型

```
type typeName func(input1 inputType1 , input2 inputType2 [, ...]) (
```

函数作为类型到底有什么好处呢？那就是可以把这个类型的函数当做值来传递，请看下面的例子

```
package main
import "fmt"

type testInt func(int) bool // 声明了一个函数类型

func isOdd(integer int) bool {
    if integer%2 == 0 {
        return false
    }
    return true
}

func isEven(integer int) bool {
    if integer%2 == 0 {
        return true
    }
    return false
}

// 声明的函数类型在这个地方当做了一个参数

func filter(slice []int, f testInt) []int {
    var result []int
    for _, value := range slice {
        if f(value) {
            result = append(result, value)
        }
    }
    return result
}

func main(){
    slice := []int {1, 2, 3, 4, 5, 7}
    fmt.Println("slice = ", slice)
    odd := filter(slice, isOdd)    // 函数当做值来传递了
    fmt.Println("Odd elements of slice are: ", odd)
    even := filter(slice, isEven)  // 函数当做值来传递了
    fmt.Println("Even elements of slice are: ", even)
}
```

函数当做值和类型在我们写一些通用接口的时候非常有用，通过上面例子我们看到 `testInt` 这个类型是一个函数类型，然后两个 `filter` 函数的参数和返回值与 `testInt` 类型是一样的，但是我们可以实现很多种的逻辑，这样使得我们的程

序变得非常的灵活。

Panic和Recover

Go没有像Java那样的异常机制，它不能抛出异常，而是使用了 `panic` 和 `recover` 机制。一定要记住，你应当把它作为最后的手段来使用，也就是说，你的代码中应当没有，或者很少有 `panic` 的东西。这是个强大的工具，请明智地使用它。那么，我们应该如何使用它呢？

Panic

> 是一个内建函数，可以中断原有的控制流程，进入一个令人恐慌的流程中。当函数 `F` 调用 `panic` ，函数F的执行被中断，但是 `F` 中的延迟函数会正常执行，然后F返回到调用它的地方。在调用的地方， `F` 的行为就像调用了 `panic` 。这一过程继续向上，直到发生 `panic` 的 `goroutine` 中所有调用的函数返回，此时程序退出。恐慌可以直接调用 `panic` 产生。也可以由运行时错误产生，例如访问越界的数组。

Recover

> 是一个内建的函数，可以让进入令人恐慌的流程中的 `goroutine` 恢复过来。 `recover` 仅在延迟函数中有效。在正常的执行过程中，调用 `recover` 会返回 `nil` ，并且没有其它任何效果。如果当前的 `goroutine` 陷入恐慌，调用 `recover` 可以捕获到 `panic` 的输入值，并且恢复正常的执行。

下面这个函数演示了如何在过程中使用 `panic`

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

下面这个函数检查作为其参数的函数在执行时是否会产生 `panic` ：

```
func throwsPanic(f func()) (b bool) {
    defer func() {
        if x := recover(); x != nil {
            b = true
        }
    }()
    f() //执行函数f，如果f中出现了panic，那么就可以恢复回来
    return
}
```

`main` 函数和 `init` 函数

Go里面有两个保留的函数： `init` 函数（能够应用于所有的 `package` ）和 `main` 函数（只能应用于 `package main` ）。这两个函数在定义时不能有任何的参数和返回值。虽然一个 `package` 里面可以写任意多个 `init` 函数，但这无论是对于可读性还是以后的可维护性来说，我们都强烈建议用户在一个 `package` 中每个文件只写一个 `init` 函数。

Go程序会自动调用 `init()` 和 `main()` ，所以你不需要在任何地方调用这两个函数。每个 `package` 中的 `init` 函数都是可选的，但 `package main` 就必须包含一个 `main` 函数。

程序的初始化和执行都起始于 `main` 包。如果 `main` 包还导入了其它的包，那么就会在编译时将它们依次导入。有时一个包会被多个包同时导入，那么它只会被导入一次（例如很多包可能都会用到 `fmt` 包，但它只会被导入一次，因为没有必要导入多次）。当一个包被导入时，如果该包还导入了其它的包，那么会先将其它包导入进来，然后再对这些包中的包级常量和变量进行初始化，接着执行 `init` 函数（如果有的话），依次类推。等所有被导入的包都加载完毕了，就会开始对 `main` 包中的包级常量和变量进行初始化，然后执行 `main` 包中的 `init` 函数（如果存在的话），最后执行 `main` 函数。下图详细地解释了整个执行过程：

图2.6 main函数引入包初始化流程图

import

我们在写Go代码的时候经常用到import这个命令用来导入包文件，而我们经常看到的方式参考如下：

```
import(
    "fmt"
)
```

然后我们代码里面可以通过如下的方式调用

```
fmt.Println("hello world")
```

上面这个fmt是Go语言的标准库，其实是去 `GOROOT` 环境变量指定目录下去加载该模块，当然Go的import还支持如下两种方式来加载自己写的模块：

1. 相对路径

   import “./model” //当前文件同一目录的model目录，但是不建议这种方式来import

2. 绝对路径

import “shorturl/model” //加载gopath/src/shorturl/model模块

上面展示了一些import常用的几种方式，但是还有一些特殊的import，让很多新手很费解，下面我们来一一讲解一下到底是怎么一回事

1. 点操作

   我们有时候会看到如下的方式导入包

   ```
   import(
       . "fmt"
   )
   ```

   这个点操作的含义就是这个包导入之后在你调用这个包的函数时，你可以省略前缀的包名，也就是前面你调用的fmt.Println("hello world")可以省略的写成Println("hello world")

2. 别名操作

   别名操作顾名思义我们可以把包命名成另一个我们用起来容易记忆的名字

   ```
   import(
       f "fmt"
   )
   ```

   别名操作的话调用包函数时前缀变成了我们的前缀，即f.Println("hello world")

3. _操作

   这个操作经常是让很多人费解的一个操作符，请看下面这个import

   ```
   import (
       "database/sql"
       _ "github.com/ziutek/mymysql/godrv"
   )
   ```

   _操作其实是引入该包，而不直接使用包里面的函数，而是调用了该包里面的init函数。

links

- 目录
- 上一章: Go基础
- 下一节: struct类型

2.4 struct类型

struct

Go语言中，也和C或者其他语言一样，我们可以声明新的类型，作为其它类型的属性或字段的容器。例如，我们可以创建一个自定义类型 `person` 代表一个人的实体。这个实体拥有属性：姓名和年龄。这样的类型我们称之 `struct` 。如下代码所示:

```
type person struct {
    name string
    age int
}
```

看到了吗？声明一个struct如此简单，上面的类型包含有两个字段

- 一个string类型的字段name，用来保存用户名称这个属性
- 一个int类型的字段age,用来保存用户年龄这个属性

如何使用struct呢？请看下面的代码

```
type person struct {
    name string
    age int
}

var P person  // P现在就是person类型的变量了

P.name = "Astaxie"  // 赋值"Astaxie"给P的name属性.
P.age = 25  // 赋值"25"给变量P的age属性
fmt.Printf("The person's name is %s", P.name)  // 访问P的name属性.
```

除了上面这种P的声明使用之外，还有另外几种声明使用方式：

- 1.按照顺序提供初始化值

  P := person{"Tom", 25}

- 2.通过 `field:value` 的方式初始化，这样可以任意顺序

  P := person{age:24, name:"Tom"}

- 3.当然也可以通过 `new` 函数分配一个指针，此处P的类型为*person

  P := new(person)

下面我们看一个完整的使用struct的例子

```
package main
import "fmt"

// 声明一个新的类型
type person struct {
    name string
    age int
}

// 比较两个人的年龄，返回年龄大的那个人，并且返回年龄差
// struct也是传值的
func Older(p1, p2 person) (person, int) {
    if p1.age>p2.age {  // 比较p1和p2这两个人的年龄
        return p1, p1.age-p2.age
    }
    return p2, p2.age-p1.age
}

func main() {
    var tom person

    // 赋值初始化
    tom.name, tom.age = "Tom", 18

    // 两个字段都写清楚的初始化
    bob := person{age:25, name:"Bob"}

    // 按照struct定义顺序初始化值
    paul := person{"Paul", 43}

    tb_Older, tb_diff := Older(tom, bob)
    tp_Older, tp_diff := Older(tom, paul)
    bp_Older, bp_diff := Older(bob, paul)

    fmt.Printf("Of %s and %s, %s is older by %d years\n",
        tom.name, bob.name, tb_Older.name, tb_diff)

    fmt.Printf("Of %s and %s, %s is older by %d years\n",
        tom.name, paul.name, tp_Older.name, tp_diff)

    fmt.Printf("Of %s and %s, %s is older by %d years\n",
        bob.name, paul.name, bp_Older.name, bp_diff)
}
```

struct的匿名字段

我们上面介绍了如何定义一个struct，定义的时候是字段名与其类型一一对应，实际上Go支持只提供类型，而不写字段名的方式，也就是匿名字段，也称为嵌入字段。

当匿名字段是一个struct的时候，那么这个struct所拥有的全部字段都被隐式地引入了当前定义的这个struct。

让我们来看一个例子，让上面说的这些更具体化

```
package main
import "fmt"

type Human struct {
    name string
    age int
    weight int
}

type Student struct {
    Human  // 匿名字段，那么默认Student就包含了Human的所有字段
    speciality string
}

func main() {
    // 我们初始化一个学生
    mark := Student{Human{"Mark", 25, 120}, "Computer Science"}

    // 我们访问相应的字段
    fmt.Println("His name is ", mark.name)
    fmt.Println("His age is ", mark.age)
    fmt.Println("His weight is ", mark.weight)
    fmt.Println("His speciality is ", mark.speciality)
    // 修改对应的备注信息
    mark.speciality = "AI"
    fmt.Println("Mark changed his speciality")
    fmt.Println("His speciality is ", mark.speciality)
    // 修改他的年龄信息
    fmt.Println("Mark become old")
    mark.age = 46
    fmt.Println("His age is", mark.age)
    // 修改他的体重信息
    fmt.Println("Mark is not an athlet anymore")
    mark.weight += 60
    fmt.Println("His weight is", mark.weight)
}
```

图例如下:

图2.7 Student和Human的方法继承

我们看到Student访问属性age和name的时候，就像访问自己所有用的字段一样，对，匿名字段就是这样，能够实现字段的继承。是不是很酷啊？还有比这个更酷的呢，那就是student还能访问Human这个字段作为字段名。请看下面的代码，是不是更酷了。

```
mark.Human = Human{"Marcus", 55, 220}
mark.Human.age -= 1
```

通过匿名访问和修改字段相当的有用，但是不仅仅是struct字段哦，所有的内置类型和自定义类型都是可以作为匿名字段的。请看下面的例子

```
package main
import "fmt"

type Skills []string

type Human struct {
    name string
    age int
    weight int
}

type Student struct {
    Human  // 匿名字段，struct
    Skills // 匿名字段，自定义的类型string slice
    int    // 内置类型作为匿名字段
    speciality string
}

func main() {
    // 初始化学生Jane
    jane := Student{Human:Human{"Jane", 35, 100}, speciality:"Biolo
    // 现在我们来访问相应的字段
    fmt.Println("Her name is ", jane.name)
    fmt.Println("Her age is ", jane.age)
    fmt.Println("Her weight is ", jane.weight)
    fmt.Println("Her speciality is ", jane.speciality)
    // 我们来修改他的skill技能字段
    jane.Skills = []string{"anatomy"}
    fmt.Println("Her skills are ", jane.Skills)
    fmt.Println("She acquired two new ones ")
    jane.Skills = append(jane.Skills, "physics", "golang")
    fmt.Println("Her skills now are ", jane.Skills)
    // 修改匿名内置类型字段
    jane.int = 3
    fmt.Println("Her preferred number is", jane.int)
}
```

从上面例子我们看出来struct不仅仅能够将struct作为匿名字段、自定义类型、内置类型都可以作为匿名字段，而且可以在相应的字段上面进行函数操作（如例子中的append）。

这里有一个问题：如果human里面有一个字段叫做phone，而student也有一个字段叫做phone，那么该怎么办呢？

Go里面很简单的解决了这个问题，最外层的优先访问，也就是当你通过 `student.phone` 访问的时候，是访问student里面的字段，而不是human里面的字段。

这样就允许我们去重载通过匿名字段继承的一些字段，当然如果我们想访问重载后对应匿名类型里面的字段，可以通过匿名字段名来访问。请看下面的例子

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string  // Human类型拥有的字段
}

type Employee struct {
    Human  // 匿名字段Human
    speciality string
    phone string  // 雇员的phone字段
}

func main() {
    Bob := Employee{Human{"Bob", 34, "777-444-XXXX"}, "Designer", '
    fmt.Println("Bob's work phone is:", Bob.phone)
    // 如果我们要访问Human的phone字段
    fmt.Println("Bob's personal phone is:", Bob.Human.phone)
}
```

links

- 目录
- 上一章: 流程和函数
- 下一节: 面向对象

2.5 面向对象

前面两章我们介绍了函数和struct，那你是否想过函数当作struct的字段一样来处理呢？今天我们就讲解一下函数的另一种形态，带有接收者的函数，我们称为 `method`

method

现在假设有这么一个场景，你定义了一个struct叫做长方形，你现在想要计算他的面积，那么按照我们一般的思路应该会用下面的方式来实现

```
package main
import "fmt"

type Rectangle struct {
    width, height float64
}

func area(r Rectangle) float64 {
    return r.width*r.height
}

func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    fmt.Println("Area of r1 is: ", area(r1))
    fmt.Println("Area of r2 is: ", area(r2))
}
```

这段代码可以计算出来长方形的面积，但是area()不是作为Rectangle的方法实现的（类似面向对象里面的方法），而是将Rectangle的对象（如r1,r2）作为参数传入函数计算面积的。

这样实现当然没有问题咯，但是当需要增加圆形、正方形、五边形甚至其它多边形的时候，你想计算他们的面积的时候怎么办啊？那就只能增加新的函数咯，但是函数名你就必须要跟着换了，变成 `area_rectangle, area_circle, area_triangle...`

像下图所表示的那样， 椭圆代表函数, 而这些函数并不从属于struct(或者以面向对象的术语来说，并不属于class)，他们是单独存在于struct外围，而非在概念上属于某个struct的。

图2.8 方法和struct的关系图

很显然，这样的实现并不优雅，并且从概念上来说"面积"是"形状"的一个属性，它是属于这个特定的形状的，就像长方形的长和宽一样。

基于上面的原因所以就有了 `method` 的概念， `method` 是附属在一个给定的类型上的，他的语法和函数的声明语法几乎一样，只是在 `func` 后面增加了一个receiver(也就是method所依从的主体)。

用上面提到的形状的例子来说，method `area()` 是依赖于某个形状(比如说Rectangle)来发生作用的。Rectangle.area()的发出者是Rectangle， area()是属于Rectangle的方法，而非一个外围函数。

更具体地说，Rectangle存在字段length 和 width, 同时存在方法area(), 这些字段和方法都属于Rectangle。

用Rob Pike的话来说就是：

> "A method is a function with an implicit first argument, called a receiver."

method的语法如下：

```
func (r ReceiverType) funcName(parameters) (results)
```

下面我们用最开始的例子用method来实现：

```
package main
import (
    "fmt"
    "math"
)

type Rectangle struct {
    width, height float64
}

type Circle struct {
    radius float64
}

func (r Rectangle) area() float64 {
    return r.width*r.height
}

func (c Circle) area() float64 {
    return c.radius * c.radius * math.Pi
}


func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    c1 := Circle{10}
    c2 := Circle{25}

    fmt.Println("Area of r1 is: ", r1.area())
    fmt.Println("Area of r2 is: ", r2.area())
    fmt.Println("Area of c1 is: ", c1.area())
    fmt.Println("Area of c2 is: ", c2.area())
}
```

在使用method的时候重要注意几点

- 虽然method的名字一模一样，但是如果接收者不一样，那么method就不一样
- method里面可以访问接收者的字段
- 调用method通过 `.` 访问，就像struct里面访问字段一样

图示如下:

图2.9 不同struct的method不同

在上例，method area() 分别属于Rectangle和Circle， 于是他们的 Receiver 就变成了Rectangle 和 Circle, 或者说，这个area()方法 是由 Rectangle/Circle 发出的。

> 值得说明的一点是，图示中method用虚线标出，意思是此处方法的Receiver是以值传递，而非引用传递，是的，Receiver还可以是指针, 两者的差别在于, 指针作为Receiver会对实例对象的内容发生操作,而普通类型作为Receiver仅仅是以副本作为操作对象,并不对原实例对象发生操作。后文对此会有详细论述。

那是不是method只能作用在struct上面呢？当然不是咯，他可以定义在任何你自定义的类型、内置类型、struct等各种类型上面。这里你是不是有点迷糊了，什么叫自定义类型，自定义类型不就是struct嘛，不是这样的哦，struct只是自定义类型里面一种比较特殊的类型而已，还有其他自定义类型申明，可以通过如下这样的申明来实现。

```
type typeName typeLiteral
```

请看下面这个申明自定义类型的代码

```
type ages int

type money float32

type months map[string]int

m := months {
    "January":31,
    "February":28,
    ...
    "December":31,
}
```

看到了吗？简单的很吧，这样你就可以在自己的代码里面定义有意义的类型了，实际上只是一个定义了一个别名,有点类似于c中的typedef，例如上面ages替代了int

好了，让我们回到 `method`

你可以在任何的自定义类型中定义任意多的 `method` ，接下来让我们看一个复杂一点的例子

```
package main
import "fmt"

const(
    WHITE = iota
    BLACK
    BLUE
    RED
    YELLOW
)

type Color byte
```

```
type Box struct {
    width, height, depth float64
    color Color
}

type BoxList []Box //a slice of boxes

func (b Box) Volume() float64 {
    return b.width * b.height * b.depth
}

func (b *Box) SetColor(c Color) {
    b.color = c
}

func (bl BoxList) BiggestColor() Color {
    v := 0.00
    k := Color(WHITE)
    for _, b := range bl {
        if bv := b.Volume(); bv > v {
            v = bv
            k = b.color
        }
    }
    return k
}

func (bl BoxList) PaintItBlack() {
    for i, _ := range bl {
        bl[i].SetColor(BLACK)
    }
}

func (c Color) String() string {
    strings := []string {"WHITE", "BLACK", "BLUE", "RED", "YELLOW"]
    return strings[c]
}

func main() {
    boxes := BoxList {
        Box{4, 4, 4, RED},
        Box{10, 10, 1, YELLOW},
        Box{1, 1, 20, BLACK},
        Box{10, 10, 1, BLUE},
        Box{10, 30, 1, WHITE},
        Box{20, 20, 20, YELLOW},
    }

    fmt.Printf("We have %d boxes in our set\n", len(boxes))
    fmt.Println("The volume of the first one is", boxes[0].Volume()
    fmt.Println("The color of the last one is",boxes[len(boxes)-1].
    fmt.Println("The biggest one is", boxes.BiggestColor().String()
```

```
    fmt.Println("Let's paint them all black")
    boxes.PaintItBlack()
    fmt.Println("The color of the second one is", boxes[1].color.St

    fmt.Println("Obviously, now, the biggest one is", boxes.Biggest
}
```

上面的代码通过const定义了一些常量，然后定义了一些自定义类型

- Color作为byte的别名
- 定义了一个struct:Box，含有三个长宽高字段和一个颜色属性
- 定义了一个slice:BoxList，含有Box

然后以上面的自定义类型为接收者定义了一些method

- Volume()定义了接收者为Box，返回Box的容量
- SetColor(c Color)，把Box的颜色改为c
- BiggestColor()定在在BoxList上面，返回list里面容量最大的颜色
- PaintItBlack()把BoxList里面所有Box的颜色全部变成黑色
- String()定义在Color上面，返回Color的具体颜色(字符串格式)

上面的代码通过文字描述出来之后是不是很简单？我们一般解决问题都是通过问题的描述，去写相应的代码实现。

指针作为receiver

现在让我们回过头来看看SetColor这个method，它的receiver是一个指向Box的指针，是的，你可以使用*Box。想想为啥要使用指针而不是Box本身呢？

我们定义SetColor的真正目的是想改变这个Box的颜色，如果不传Box的指针，那么SetColor接受的其实是Box的一个copy，也就是说method内对于颜色值的修改，其实只作用于Box的copy，而不是真正的Box。所以我们需要传入指针。

这里可以把receiver当作method的第一个参数来看，然后结合前面函数讲解的传值和传引用就不难理解

这里你也许会问了那SetColor函数里面应该这样定义 `*b.Color=c` ,而不是 `b.Color=c` ,因为我们需要读取到指针相应的值。

你是对的，其实Go里面这两种方式都是正确的，当你用指针去访问相应的字段时(虽然指针没有任何的字段)，Go知道你要通过指针去获取这个值，看到了吧，Go的设计是不是越来越吸引你了。

也许细心的读者会问这样的问题，PaintItBlack里面调用SetColor的时候是不是应该写成 `(&bl[i]).SetColor(BLACK)` ，因为SetColor的receiver是*Box，而不是Box。

你又说对的，这两种方式都可以，因为Go知道receiver是指针，他自动帮你转了。

也就是说：

> 如果一个method的receiver是*T,你可以在一个T类型的实例变量V上面调用这个method，而不需要&V去调用这个method

类似的

> 如果一个method的receiver是T，你可以在一个*T*类型的变量*P*上面调用这个*method*，而不需要 P去调用这个method

所以，你不用担心你是调用的指针的method还是不是指针的method，Go知道你要做的一切，这对于有多年C/C++编程经验的同学来说，真是解决了一个很大的痛苦。

method继承

前面一章我们学习了字段的继承，那么你也会发现Go的一个神奇之处，method也是可以继承的。如果匿名字段实现了一个method，那么包含这个匿名字段的struct也能调用该method。让我们来看下面这个例子

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human //匿名字段
    school string
}

type Employee struct {
    Human //匿名字段
    company string
}

//在human上面定义了一个method
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.pho
}

func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"]

    mark.SayHi()
    sam.SayHi()
}
```

method重写

上面的例子中，如果Employee想要实现自己的SayHi,怎么办？简单，和匿名字段冲突一样的道理，我们可以在Employee上面定义一个method，重写了匿名字段的方法。请看下面的例子

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human //匿名字段
    school string
}

type Employee struct {
    Human //匿名字段
    company string
}

//Human定义method
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.pho
}

//Employee的method重写Human的method
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name
        e.company, e.phone) //Yes you can split into 2 lines here.
}

func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}

    mark.SayHi()
    sam.SayHi()
}
```

上面的代码设计的是如此的美妙，让人不自觉的为Go的设计惊叹！

通过这些内容，我们可以设计出基本的面向对象的程序了，但是Go里面的面向对象是如此的简单，没有任何的私有、公有关键字，通过大小写来实现(大写开头的为公有，小写开头的为私有)，方法也同样适用这个原则。

links

- 目录
- 上一章: struct类型

- 下一节: interface

2.6 interface

interface

Go语言里面设计最精妙的应该算interface，它让面向对象，内容组织实现非常的方便，当你看完这一章，你就会被interface的巧妙设计所折服。

什么是interface

简单的说，interface是一组method的组合，我们通过interface来定义对象的一组行为。

我们前面一章最后一个例子中Student和Employee都能SayHi，虽然他们的内部实现不一样，但是那不重要，重要的是他们都能 `say hi`

让我们来继续做更多的扩展，Student和Employee实现另一个方法 `Sing` ，然后Student实现方法BorrowMoney而Employee实现SpendSalary。

这样Student实现了三个方法：SayHi、Sing、BorrowMoney；而Employee实现了SayHi、Sing、SpendSalary。

上面这些方法的组合称为interface(被对象Student和Employee实现)。例如Student和Employee都实现了interface：SayHi和Sing，也就是这两个对象是该interface类型。而Employee没有实现这个interface：SayHi、Sing和BorrowMoney，因为Employee没有实现BorrowMoney这个方法。

interface类型

interface类型定义了一组方法，如果某个对象实现了某个接口的所有方法，则此对象就实现了此接口。详细的语法参考下面这个例子

```
type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human //匿名字段Human
    school string
    loan float32
}

type Employee struct {
    Human //匿名字段Human
    company string
```

```
    money float32
}

//Human对象实现Sayhi方法
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.pho
}

// Human对象实现Sing方法
func (h *Human) Sing(lyrics string) {
    fmt.Println("La la, la la la, la la la la la...", lyrics)
}

//Human对象实现Guzzle方法
func (h *Human) Guzzle(beerStein string) {
    fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}

// Employee重载Human的Sayhi方法
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name
        e.company, e.phone) //此句可以分成多行
}

//Student实现BorrowMoney方法
func (s *Student) BorrowMoney(amount float32) {
    s.loan += amount // (again and again and...)
}

//Employee实现SpendSalary方法
func (e *Employee) SpendSalary(amount float32) {
    e.money -= amount // More vodka please!!! Get me through the da
}

// 定义interface
type Men interface {
    SayHi()
    Sing(lyrics string)
    Guzzle(beerStein string)
}

type YoungChap interface {
    SayHi()
    Sing(song string)
    BorrowMoney(amount float32)
}

type ElderlyGent interface {
    SayHi()
    Sing(song string)
    SpendSalary(amount float32)
}
```

通过上面的代码我们可以知道，interface可以被任意的对象实现。我们看到上面的Men interface被Human、Student和Employee实现。同理，一个对象可以实现任意多个interface，例如上面的Student实现了Men和YoungChap两个interface。

最后，任意的类型都实现了空interface(我们这样定义：interface{})，也就是包含0个method的interface。

interface值

那么interface里面到底能存什么值呢？如果我们定义了一个interface的变量，那么这个变量里面可以存实现这个interface的任意类型的对象。例如上面例子中，我们定义了一个Men interface类型的变量m，那么m里面可以存Human、Student或者Employee值。

因为m能够持有这三种类型的对象，所以我们可以定义一个包含Men类型元素的slice，这个slice可以被赋予实现了Men接口的任意结构的对象，这个和我们传统意义上面的slice有所不同。

让我们来看一下下面这个例子:

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human //匿名字段
    school string
    loan float32
}

type Employee struct {
    Human //匿名字段
    company string
    money float32
}

//Human实现SayHi方法
func (h Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.pho
}

//Human实现Sing方法
func (h Human) Sing(lyrics string) {
    fmt.Println("La la la la...", lyrics)
}
```

```
//Employee重载Human的SayHi方法
func (e Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name
        e.company, e.phone)
    }

// Interface Men被Human,Student和Employee实现
// 因为这三个类型都实现了这两个方法
type Men interface {
    SayHi()
    Sing(lyrics string)
}

func main() {
    mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}
    paul := Student{Human{"Paul", 26, "111-222-XXX"}, "Harvard", 10
    sam := Employee{Human{"Sam", 36, "444-222-XXX"}, "Golang Inc.",
    tom := Employee{Human{"Tom", 37, "222-444-XXX"}, "Things Ltd.",

    //定义Men类型的变量i
    var i Men

    //i能存储Student
    i = mike
    fmt.Println("This is Mike, a Student:")
    i.SayHi()
    i.Sing("November rain")

    //i也能存储Employee
    i = tom
    fmt.Println("This is tom, an Employee:")
    i.SayHi()
    i.Sing("Born to be wild")

    //定义了slice Men
    fmt.Println("Let's use a slice of Men and see what happens")
    x := make([]Men, 3)
    //这三个都是不同类型的元素，但是他们实现了interface同一个接口
    x[0], x[1], x[2] = paul, sam, mike

    for _, value := range x{
        value.SayHi()
    }
}
```

通过上面的代码，你会发现interface就是一组抽象方法的集合，它必须由其他非interface类型实现，而不能自我实现， Go通过interface实现了duck-typing:即"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子"。

空interface

空interface(interface{})不包含任何的method，正因为如此，所有的类型都实现了空interface。空interface对于描述起不到任何的作用(因为它不包含任何的method），但是空interface在我们需要存储任意类型的数值的时候相当有用，因为它可以存储任意类型的数值。它有点类似于C语言的void*类型。

```
// 定义a为空接口
var a interface{}
var i int = 5
s := "Hello world"
// a可以存储任意类型的数值
a = i
a = s
```

一个函数把interface{}作为参数，那么他可以接受任意类型的值作为参数，如果一个函数返回interface{},那么也就可以返回任意类型的值。是不是很有用啊！

interface函数参数

interface的变量可以持有任意实现该interface类型的对象，这给我们编写函数(包括method)提供了一些额外的思考，我们是不是可以通过定义interface参数，让函数接受各种类型的参数。

举个例子：fmt.Println是我们常用的一个函数，但是你是否注意到它可以接受任意类型的数据。打开fmt的源码文件，你会看到这样一个定义:

```
type Stringer interface {
     String() string
}
```

也就是说，任何实现了String方法的类型都能作为参数被fmt.Println调用,让我们来试一试

```
package main
import (
    "fmt"
    "strconv"
)

type Human struct {
    name string
    age int
    phone string
}

// 通过这个方法 Human 实现了 fmt.Stringer
func (h Human) String() string {
    return "❰"+h.name+" - "+strconv.Itoa(h.age)+" years -  ✆ " +h.p
}

func main() {
    Bob := Human{"Bob", 39, "000-7777-XXX"}
    fmt.Println("This Human is : ", Bob)
}
```

现在我们再回顾一下前面的Box示例，你会发现Color结构也定义了一个method：String。其实这也是实现了fmt.Stringer这个interface，即如果需要某个类型能被fmt包以特殊的格式输出，你就必须实现Stringer这个接口。如果没有实现这个接口，fmt将以默认的方式输出。

```
//实现同样的功能
fmt.Println("The biggest one is", boxes.BiggestsColor().String())
fmt.Println("The biggest one is", boxes.BiggestsColor())
```

注：实现了error接口的对象（即实现了Error() string的对象），使用fmt输出时，会调用Error()方法，因此不必再定义String()方法了。

interface变量存储的类型

我们知道interface的变量里面可以存储任意类型的数值(该类型实现了interface)。那么我们怎么反向知道这个变量里面实际保存了的是哪个类型的对象呢？目前常用的有两种方法：

- Comma-ok断言

  Go语言里面有一个语法，可以直接判断是否是该类型的变量： value, ok = element.(T)，这里value就是变量的值，ok是一个bool类型，element是interface变量，T是断言的类型。

如果element里面确实存储了T类型的数值，那么ok返回true，否则返回false。

让我们通过一个例子来更加深入的理解。

```
package main

import (
    "fmt"
    "strconv"
)

type Element interface{}
type List [] Element

type Person struct {
    name string
    age int
}

//定义了String方法，实现了fmt.Stringer
func (p Person) String() string {
    return "(name: " + p.name + " - age: "+strconv.Itoa(p.age
}

func main() {
    list := make(List, 3)
    list[0] = 1 // an int
    list[1] = "Hello" // a string
    list[2] = Person{"Dennis", 70}

    for index, element := range list {
        if value, ok := element.(int); ok {
            fmt.Printf("list[%d] is an int and its value is %
        } else if value, ok := element.(string); ok {
            fmt.Printf("list[%d] is a string and its value is
        } else if value, ok := element.(Person); ok {
            fmt.Printf("list[%d] is a Person and its value is
        } else {
            fmt.Printf("list[%d] is of a different type\n", i
        }
    }
}
```

是不是很简单啊，同时你是否注意到了多个if里面，还记得我前面介绍流程时讲过，if里面允许初始化变量。

也许你注意到了，我们断言的类型越多，那么if else也就越多，所以才引出了下面要介绍的switch。

- switch测试

最好的讲解就是代码例子，现在让我们重写上面的这个实现

```
package main

import (
    "fmt"
    "strconv"
)

type Element interface{}
type List [] Element

type Person struct {
    name string
    age int
}

//打印
func (p Person) String() string {
    return "(name: " + p.name + " - age: "+strconv.Itoa(p.age
}

func main() {
    list := make(List, 3)
    list[0] = 1 //an int
    list[1] = "Hello" //a string
    list[2] = Person{"Dennis", 70}

    for index, element := range list{
        switch value := element.(type) {
            case int:
                fmt.Printf("list[%d] is an int and its value
            case string:
                fmt.Printf("list[%d] is a string and its valu
            case Person:
                fmt.Printf("list[%d] is a Person and its valu
            default:
                fmt.Println("list[%d] is of a different type"
        }
    }
}
```

这里有一点需要强调的是：`element.(type)` 语法不能在switch外的任何逻辑里面使用，如果你要在switch外面判断一个类型就使用 `comma-ok` 。

嵌入interface

Go里面真正吸引人的是它内置的逻辑语法，就像我们在学习Struct时学习的匿名字段，多么的优雅啊，那么相同的逻辑引入到interface里面，那不是更加完美了。如果一个interface1作为interface2的一个嵌入字段，那么interface2隐式的包含了interface1里面的method。

我们可以看到源码包container/heap里面有这样的一个定义

```
type Interface interface {
    sort.Interface //嵌入字段sort.Interface
    Push(x interface{}) //a Push method to push elements into the h
    Pop() interface{} //a Pop elements that pops elements from the
}
```

我们看到sort.Interface其实就是嵌入字段，把sort.Interface的所有method给隐式的包含进来了。也就是下面三个方法：

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less returns whether the element with index i should sort
    // before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

另一个例子就是io包下面的 io.ReadWriter，它包含了io包下面的Reader和Writer两个interface：

```
// io.ReadWriter
type ReadWriter interface {
    Reader
    Writer
}
```

反射

Go语言实现了反射，所谓反射就是能检查程序在运行时的状态。我们一般用到的包是reflect包。如何运用reflect包，官方的这篇文章详细的讲解了reflect包的实现原理，laws of reflection

使用reflect一般分成三步，下面简要的讲解一下：要去反射是一个类型的值(这些值都实现了空interface)，首先需要把它转化成reflect对象(reflect.Type或者reflect.Value，根据不同的情况调用不同的函数)。这两种获取方式如下：

```
t := reflect.TypeOf(i)    //得到类型的元数据,通过t我们能获取类型定义里面的
v := reflect.ValueOf(i)   //得到实际的值，通过v我们获取存储在里面的值，还可
```

转化为reflect对象之后我们就可以进行一些操作了，也就是将reflect对象转化成相应的值，例如

```
tag := t.Elem().Field(0).Tag  //获取定义在struct里面的标签
name := v.Elem().Field(0).String()  //获取存储在第一个字段里面的值
```

获取反射值能返回相应的类型和数值

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
fmt.Println("value:", v.Float())
```

最后，反射的话，那么反射的字段必须是可修改的，我们前面学习过传值和传引用，这个里面也是一样的道理。反射的字段必须是可读写的意思是，如果下面这样写，那么会发生错误

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1)
```

如果要修改相应的值，必须这样写

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
v.SetFloat(7.1)
```

上面只是对反射的简单介绍，更深入的理解还需要自己在编程中不断的实践。

links

- 目录
- 上一章: 面向对象
- 下一节: 并发

2.7 并发

有人把Go比作21世纪的C语言，第一是因为Go语言设计简单，第二，21世纪最重要的就是并行程序设计，而Go从语言层面就支持了并行。

goroutine

goroutine是Go并行设计的核心。goroutine说到底其实就是线程，但是它比线程更小，十几个goroutine可能体现在底层就是五六个线程，Go语言内部帮你实现了这些goroutine之间的内存共享。执行goroutine只需极少的栈内存(大概是4~5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。goroutine比thread更易用、更高效、更轻便。

goroutine是通过Go的runtime管理的一个线程管理器。goroutine通过`go`关键字实现了，其实就是一个普通的函数。

```
go hello(a, b, c)
```

通过关键字go就启动了一个goroutine。我们来看一个例子

```
package main

import (
    "fmt"
    "runtime"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        runtime.Gosched()
        fmt.Println(s)
    }
}

func main() {
    go say("world") //开一个新的Goroutines执行
    say("hello") //当前Goroutines执行
}

// 以上程序执行后将输出：
// hello
// world
// hello
// world
// hello
// world
// hello
// world
// hello
```

我们可以看到go关键字很方便的就实现了并发编程。 上面的多个goroutine运行在同一个进程里面，共享内存数据，不过设计上我们要遵循：不要通过共享来通信，而要通过通信来共享。

> runtime.Gosched()表示让CPU把时间片让给别人,下次某个时候继续恢复执行该goroutine。
>
> 默认情况下，调度器仅使用单线程，也就是说只实现了并发。想要发挥多核处理器的并行，需要在我们的程序中显式调用 runtime.GOMAXPROCS(n) 告诉调度器同时使用多个线程。GOMAXPROCS 设置了同时运行逻辑代码的系统线程的最大数量，并返回之前的设置。如果n < 1，不会改变当前设置。以后Go的新版本中调度得到改进后，这将被移除。这里有一篇Rob介绍的关于并发和并行的文章：http://concur.rspace.googlecode.com/hg/talk/concur.html#landing-slide

channels

goroutine运行在相同的地址空间，因此访问共享内存必须做好同步。那么goroutine之间如何进行数据的通信呢，Go提供了一个很好的通信机制channel。channel可以与Unix shell 中的双向管道做类比：可以通过它发送或者接收值。这些值只能是特定的类型：channel类型。定义一个channel时，也需要定义发送到channel的值的类型。注意，必须使用make 创建channel：

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

channel通过操作符 `<-` 来接收和发送数据

```
ch <- v    // 发送v到channel ch.
v := <-ch  // 从ch中接收数据，并赋值给v
```

我们把这些应用到我们的例子中来：

```
package main

import "fmt"

func sum(a []int, c chan int) {
    total := 0
    for _, v := range a {
        total += v
    }
    c <- total  // send total to c
}

func main() {
    a := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(a[:len(a)/2], c)
    go sum(a[len(a)/2:], c)
    x, y := <-c, <-c  // receive from c

    fmt.Println(x, y, x + y)
}
```

默认情况下，channel接收和发送数据都是阻塞的，除非另一端已经准备好，这样就使得Goroutines同步变的更加的简单，而不需要显式的lock。所谓阻塞，也就是如果读取（value := <-ch）它将会被阻塞，直到有数据接收。其次，任何发送（ch<-5）将会被阻塞，直到数据被读出。无缓冲channel是在多个goroutine之间同步很棒的工具。

Buffered Channels

上面我们介绍了默认的非缓存类型的channel，不过Go也允许指定channel的缓冲大小，很简单，就是channel可以存储多少元素。ch:= make(chan bool, 4)，创建了可以存储4个元素的bool 型channel。在这个channel 中，前4个元素可以无阻塞的写入。当写入第5个元素时，代码将会阻塞，直到其他goroutine从channel 中读取一些元素，腾出空间。

```
ch := make(chan type, value)

value == 0 ! 无缓冲（阻塞）
value > 0 ! 缓冲（非阻塞，直到value 个元素）
```

我们看一下下面这个例子，你可以在自己本机测试一下，修改相应的value值

```
package main

import "fmt"

func main() {
    c := make(chan int, 2)//修改2为1就报错，修改2为3可以正常运行
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}
    //修改为1报如下的错误:
    //fatal error: all goroutines are asleep - deadlock!
```

Range和Close

上面这个例子中，我们需要读取两次c，这样不是很方便，Go考虑到了这一点，所以也可以通过range，像操作slice或者map一样操作缓存类型的channel，请看下面的例子

```
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 1, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x + y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

`for i := range c` 能够不断的读取channel里面的数据，直到该channel被显式的关闭。上面代码我们看到可以显式的关闭channel，生产者通过内置函数 `close` 关闭channel。关闭channel之后就无法再发送任何数据了，在消费方可以通过语法 `v, ok := <-ch` 测试channel是否被关闭。如果ok返回false，那么说明channel已经没有任何数据并且已经被关闭。

> 记住应该在生产者的地方关闭channel，而不是消费的地方去关闭它，这样容易引起panic
>
> 另外记住一点的就是channel不像文件之类的，不需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束range循环之类的

Select

我们上面介绍的都是只有一个channel的情况，那么如果存在多个channel的时候，我们该如何操作呢，Go里面提供了一个关键字 `select` ，通过 `select` 可以监听channel上的数据流动。

`select` 默认是阻塞的，只有当监听的channel中有发送或接收可以进行时才会运行，当多个channel都准备好的时候，select是随机的选择一个执行的。

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 1, 1
    for {
        select {
        case c <- x:
            x, y = y, x + y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

在 `select` 里面还有default语法，`select` 其实就是类似switch的功能，default就是当监听的channel都没有准备好的时候，默认执行的（select不再阻塞等待channel）。

```
select {
case i := <-c:
    // use i
default:
    // 当c阻塞的时候执行这里
}
```

超时

有时候会出现goroutine阻塞的情况，那么我们如何避免整个程序进入阻塞的情况呢？我们可以利用select来设置超时，通过如下的方式实现：

```
func main() {
    c := make(chan int)
    o := make(chan bool)
    go func() {
        for {
            select {
                case v := <- c:
                    println(v)
                case <- time.After(5 * time.Second):
                    println("timeout")
                    o <- true
                    break
            }
        }
    }()
    <- o
}
```

runtime goroutine

runtime包中有几个处理goroutine的函数：

- Goexit

  退出当前执行的goroutine，但是defer函数还会继续调用

- Gosched

  让出当前goroutine的执行权限，调度器安排其他等待的任务运行，并在下次某个时候从该位置恢复执行。

- NumCPU

  返回 CPU 核数量

- NumGoroutine

  返回正在执行和排队的任务总数

- GOMAXPROCS

  用来设置可以并行计算的CPU核数的最大值，并返回之前的值。

links

- 目录
- 上一章: interface
- 下一节: 总结

2.8 总结

这一章我们主要介绍了Go语言的一些语法，通过语法我们可以发现Go是多么的简单，只有二十五个关键字。让我们再来回顾一下这些关键字都是用来干什么的。

```
break    default      func    interface    select
case     defer        go      map          struct
chan     else         goto    package      switch
const    fallthrough  if      range        type
continue for          import  return       var
```

- var和const参考2.2Go语言基础里面的变量和常量申明
- package和import已经有过短暂的接触
- func 用于定义函数和方法
- return 用于从函数返回
- defer 用于类似析构函数
- go 用于并发
- select 用于选择不同类型的通讯
- interface 用于定义接口，参考2.6小节
- struct 用于定义抽象数据类型，参考2.5小节
- break、case、continue、for、fallthrough、else、if、switch、goto、default这些参考2.3流程介绍里面
- chan用于channel通讯
- type用于声明自定义类型
- map用于声明map类型数据
- range用于读取slice、map、channel数据

上面这二十五个关键字记住了，那么Go你也已经差不多学会了。

links

- 目录
- 上一节: 并发
- 下一章: Web基础

3 Web基础

学习基于Web的编程可能正是你读本书的原因。事实上，如何通过Go来编写Web应用也是我编写这本书的初衷。前面已经介绍过，Go目前已经拥有了成熟的HTTP处理包，这使得编写能做任何事情的动态Web程序易如反掌。在接下来的各章中将要介绍的内容，都是属于Web编程的范畴。本章则集中讨论一些与Web相关的概念和Go如何运行Web程序的话题。

目录

links

- 目录
- 上一章: 第二章总结
- 下一节: Web工作方式

3.1 Web工作方式

我们平时浏览网页的时候,会打开浏览器，输入网址后按下回车键，然后就会显示出你想要浏览的内容。在这个看似简单的用户行为背后，到底隐藏了些什么呢？

对于普通的上网过程，系统其实是这样做的：浏览器本身是一个客户端，当你输入URL的时候，首先浏览器会去请求DNS服务器，通过DNS获取相应的域名对应的IP，然后通过IP地址找到IP对应的服务器后，要求建立TCP连接，等浏览器发送完HTTP Request（请求）包后，服务器接收到请求包之后才开始处理请求包，服务器调用自身服务，返回HTTP Response（响应）包；客户端收到来自服务器的响应后开始渲染这个Response包里的主体（body），等收到全部的内容随后断开与该服务器之间的TCP连接。

图3.1 用户访问一个Web站点的过程

一个Web服务器也被称为HTTP服务器，它通过HTTP协议与客户端通信。这个客户端通常指的是Web浏览器(其实手机端客户端内部也是浏览器实现的)。

Web服务器的工作原理可以简单地归纳为：

- 客户机通过TCP/IP协议建立到服务器的TCP连接
- 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档
- 服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端
- 客户机与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

一个简单的HTTP事务就是这样实现的，看起来很复杂，原理其实是挺简单的。需要注意的是客户机与服务器之间的通信是非持久连接的，也就是当服务器发送了应答后就与客户机断开连接，等待下一次请求。

URL和DNS解析

我们浏览网页都是通过URL访问的，那么URL到底是怎么样的呢？

URL(Uniform Resource Locator)是“统一资源定位符”的英文缩写，用于描述一个网络上的资源, 基本格式如下

```
scheme://host[:port#]/path/.../[?query-string][#anchor]
scheme         指定低层使用的协议(例如：http, https, ftp)
host           HTTP服务器的IP地址或者域名
port#          HTTP服务器的默认端口是80，这种情况下端口号可以省略。如果使用
path           访问资源的路径
query-string   发送给http服务器的数据
anchor         锚
```

DNS(Domain Name System)是“域名系统”的英文缩写，是一种组织成域层次结构的计算机和网络服务命名系统，它用于TCP/IP网络，它从事将主机名或域名转换为实际IP地址的工作。DNS就是这样的一位“翻译官”，它的基本工作原理可用下图来表示。

图3.2 DNS工作原理

更详细的DNS解析的过程如下，这个过程有助于我们理解DNS的工作模式

1. 在浏览器中输入www.qq.com域名，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

2. 如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

3. 如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP/IP参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。

4. 如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。

5. 如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置（是否设置转发器）进行查询，如果未用转发模式，本地DNS就把请求发至 “根DNS服务器”，“根DNS服务器”收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址(qq.com)给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找qq.com域服务器，重复上面的动作，进行查询，直至找到www.qq.com主机。

6. 如果用的是转发模式，此DNS服务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。

图3.3 DNS解析的整个流程

> 所谓 `递归查询过程` 就是 “查询的递交者” 更替, 而 `迭代查询过程` 则是 “查询的递交者”不变。
>
> 举个例子来说，你想知道某个一起上法律课的女孩的电话，并且你偷偷拍了她的照片，回到寝室告诉一个很仗义的哥们儿，这个哥们儿二话没说，拍着胸脯告诉你，甭急，我替你查(此处完成了一次递归查询，即，问询者的角色更替)。然后他拿着照片问了学院大四学长，学长告诉他，这姑娘是xx系的；然后这哥们儿马不停蹄又问了xx系的办公室主任助理同学，助理同学说是xx系yy班的，然后很仗义的哥们儿去xx系yy班的班长那里取到了该女孩儿电话。(此处完成若干次迭代查询，即，问询者角色不变，但反复更替问询对象)最后，他把号码交到了你手里。完成整个查询过程。

通过上面的步骤，我们最后获取的是IP地址，也就是浏览器最后发起请求的时候是基于IP来和服务器做信息交互的。

HTTP协议详解

HTTP协议是Web工作的核心，所以要了解清楚Web的工作方式就需要详细的了解清楚HTTP是怎么样工作的。

HTTP是一种让Web服务器与浏览器(客户端)通过Internet发送与接收数据的协议,它建立在TCP协议之上，一般采用TCP的80端口。它是一个请求、响应协议--客户端发出一个请求，服务器响应这个请求。在HTTP中，客户端总是通过建立一个连接与发送一个HTTP请求来发起一个事务。服务器不能主动去与客户端联系，也不能给客户端发出一个回调连接。客户端与服务器端都可以提前中断一个连接。例如，当浏览器下载一个文件时，你可以通过点击“停止”键来中断文件的下载，关闭与服务器的HTTP连接。

HTTP协议是无状态的，同一个客户端的这次请求和上次请求是没有对应关系，对HTTP服务器来说，它并不知道这两个请求是否来自同一个客户端。为了解决这个问题， Web程序引入了Cookie机制来维护连接的可持续状态。

> HTTP协议是建立在TCP协议之上的，因此TCP攻击一样会影响HTTP的通讯，例如比较常见的一些攻击：SYN Flood是当前最流行的DoS（拒绝服务攻击）与DdoS（分布式拒绝服务攻击）的方式之一，这是一种利用TCP协议缺陷，发送大量伪造的TCP连接请求，从而使得被攻击方资源耗尽（CPU满负荷或内存不足）的攻击方式。

HTTP请求包（浏览器信息）

我们先来看看Request包的结构, Request包分为3部分，第一部分叫Request line（请求行）, 第二部分叫Request header（请求头）,第三部分是body（主体）。header和body之间有个空行，请求包的例子所示:

```
GET /domains/example/ HTTP/1.1        //请求行: 请求方法 请求URI HTTP
Host：www.iana.org                //服务端的主机名
User-Agent：Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML,
Accept：text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
Accept-Encoding：gzip,deflate,sdch        //是否支持流压缩
Accept-Charset：UTF-8,*;q=0.5        //客户端字符编码集
//空行,用于分割请求头和消息体
//消息体,请求资源参数,例如POST传递的参数
```

HTTP协议定义了很多与服务器交互的请求方法，最基本的有4种，分别是GET,POST,PUT,DELETE。一个URL地址用于描述一个网络上的资源，而HTTP中的GET, POST, PUT, DELETE就对应着对这个资源的查，改，增，删4个操作。我们最常见的就是GET和POST了。GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。

通过fiddler抓包可以看到如下请求信息:

图3.4 fiddler抓取的GET信息

图3.5 fiddler抓取的POST信息

我们看看GET和POST的区别:

1. 我们可以看到GET请求消息体为空，POST请求带有消息体。
2. GET提交的数据会放在URL之后，以 `?` 分割URL和传输数据，参数之间以 `&` 相连，如 `EditPosts.aspx?name=test1&id=123456` 。POST方法是把提交的数据放在HTTP包的body中。
3. GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST方法提交的数据没有限制。
4. GET方式提交数据，会带来安全问题，比如一个登录页面，通过GET方式提交数据时，用户名和密码将出现在URL上，如果页面可以被缓存或者其他人可以访问这台机器，就可以从历史记录获得该用户的账号和密码。

HTTP响应包（服务器信息）

我们再来看看HTTP的response包，他的结构如下：

```
HTTP/1.1 200 OK                        //状态行
Server: nginx/1.0.8                    //服务器使用的WEB软件名及版本
Date:Date: Tue, 30 Oct 2012 04:14:25 GMT        //发送时间
Content-Type: text/html                //服务器发送信息的类型
Transfer-Encoding: chunked            //表示发送HTTP包是分段发的
Connection: keep-alive                //保持连接状态
Content-Length: 90                    //主体内容长度
//空行 用来分割消息头和主体
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"... /
```

Response包中的第一行叫做状态行，由HTTP协议版本号， 状态码， 状态消息 三部分组成。

状态码用来告诉HTTP客户端,HTTP服务器是否产生了预期的Response。HTTP/1.1协议中定义了5类状态码， 状态码由三位数字组成，第一个数字定义了响应的类别

- 1XX 提示信息 - 表示请求已被成功接收，继续处理
- 2XX 成功 - 表示请求已被成功接收，理解，接受
- 3XX 重定向 - 要完成请求必须进行更进一步的处理
- 4XX 客户端错误 - 请求有语法错误或请求无法实现
- 5XX 服务器端错误 - 服务器未能实现合法的请求

我们看下面这个图展示了详细的返回信息，左边可以看到有很多的资源返回码，200是常用的，表示正常信息，302表示跳转。response header里面展示了详细的信息。

图3.6 访问一次网站的全部请求信息

HTTP协议是无状态的和Connection: keep-alive的区别

无状态是指协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。从另一方面讲，打开一个服务器上的网页和你之前打开这个服务器上的网页之间没有任何联系。

HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（面对无连接）。

从HTTP/1.1起，默认都开启了Keep-Alive保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的TCP连接。

Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同服务器软件（如Apache）中设置这个时间。

请求实例

图3.7 一次请求的request和response

上面这张图我们可以了解到整个的通讯过程，同时细心的读者是否注意到了一点，一个URL请求但是左边栏里面为什么会有那么多的资源请求(这些都是静态文件，go对于静态文件有专门的处理方式)。

这个就是浏览器的一个功能，第一次请求url，服务器端返回的是html页面，然后浏览器开始渲染HTML：当解析到HTML DOM里面的图片连接，css脚本和js脚本的链接，浏览器就会自动发起一个请求静态资源的HTTP请求，获取相对应的静态资源，然后浏览器就会渲染出来，最终将所有资源整合、渲染，完整展现在我们面前的屏幕上。

> 网页优化方面有一项措施是减少HTTP请求次数，就是把尽量多的css和js资源合并在一起，目的是尽量减少网页请求静态资源的次数，提高网页加载速度，同时减缓服务器的压力。

links

- 目录
- 上一节: Web基础
- 下一节: Go搭建一个Web服务器

3.2 Go搭建一个Web服务器

前面小节已经介绍了Web是基于http协议的一个服务，Go语言里面提供了一个完善的net/http包，通过http包可以很方便的就搭建起来一个可以运行的Web服务。同时使用这个包能很简单地对Web的路由，静态文件，模版，cookie等数据进行设置和操作。

http包建立Web服务器

```
package main

import (
    "fmt"
    "net/http"
    "strings"
    "log"
)

func sayhelloName(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()  //解析参数，默认是不会解析的
    fmt.Println(r.Form)  //这些信息是输出到服务器端的打印信息
    fmt.Println("path", r.URL.Path)
    fmt.Println("scheme", r.URL.Scheme)
    fmt.Println(r.Form["url_long"])
    for k, v := range r.Form {
        fmt.Println("key:", k)
        fmt.Println("val:", strings.Join(v, ""))
    }
    fmt.Fprintf(w, "Hello astaxie!") //这个写入到w的是输出到客户端的
}

func main() {
    http.HandleFunc("/", sayhelloName) //设置访问的路由
    err := http.ListenAndServe(":9090", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

上面这个代码，我们build之后，然后执行web.exe,这个时候其实已经在9090端口监听http链接请求了。

在浏览器输入 `http://localhost:9090`

可以看到浏览器页面输出了 `Hello astaxie!`

可以换一个地址试试： `http://localhost:9090/?url_long=111&url_long=222`

看看浏览器输出的是什么，服务器输出的是什么？

在服务器端输出的信息如下：

图3.8 用户访问Web之后服务器端打印的信息

我们看到上面的代码，要编写一个Web服务器很简单，只要调用http包的两个函数就可以了。

> 如果你以前是PHP程序员，那你也许就会问，我们的nginx、apache服务器不需要吗？Go就是不需要这些，因为他直接就监听tcp端口了，做了nginx做的事情，然后sayhelloName这个其实就是我们写的逻辑函数了，跟php里面的控制层（controller）函数类似。
>
> 如果你以前是Python程序员，那么你一定听说过tornado，这个代码和他是不是很像，对，没错，Go就是拥有类似Python这样动态语言的特性，写Web应用很方便。
>
> 如果你以前是Ruby程序员，会发现和ROR的/script/server启动有点类似。

我们看到Go通过简单的几行代码就已经运行起来一个Web服务了，而且这个Web服务内部有支持高并发的特性，我将会在接下来的两个小节里面详细的讲解一下Go是如何实现Web高并发的。

links

- 目录
- 上一节: Web工作方式
- 下一节: Go如何使得web工作

3.3 Go如何使得Web工作

前面小节介绍了如何通过Go搭建一个Web服务，我们可以看到简单应用一个net/http包就方便的搭建起来了。那么Go在底层到底是怎么做的呢？万变不离其宗，Go的Web服务工作也离不开我们第一小节介绍的Web工作方式。

web工作方式的几个概念

以下均是服务器端的几个概念

Request：用户请求的信息，用来解析用户的请求信息，包括post、get、cookie、url等信息

Response：服务器需要反馈给客户端的信息

Conn：用户的每次请求链接

Handler：处理请求和生成返回信息的处理逻辑

分析http包运行机制

如下图所示，是Go实现Web服务的工作模式的流程图

图3.9 http包执行流程

1. 创建Listen Socket, 监听指定的端口, 等待客户端请求到来。
2. Listen Socket接受客户端的请求, 得到Client Socket, 接下来通过Client Socket与客户端通信。
3. 处理客户端的请求, 首先从Client Socket读取HTTP请求的协议头, 如果是POST方法, 还可能要读取客户端提交的数据, 然后交给相应的handler处理请求, handler处理完毕准备好客户端需要的数据, 通过Client Socket写给客户端。

这整个的过程里面我们只要了解清楚下面三个问题，也就知道Go是如何让Web运行起来了

- 如何监听端口？
- 如何接收客户端请求？
- 如何分配handler？

前面小节的代码里面我们可以看到，Go是通过一个函数 `ListenAndServe` 来处理这些事情的，这个底层其实这样处理的：初始化一个server对象，然后调用了 `net.Listen("tcp", addr)` ，也就是底层用TCP协议搭建了一个服务，然后监控我们设置的端口。

下面代码来自Go的http包的源码，通过下面的代码我们可以看到整个的http处理过程：

```
func (srv *Server) Serve(l net.Listener) error {
    defer l.Close()
    var tempDelay time.Duration // how long to sleep on accept fail
    for {
        rw, e := l.Accept()
        if e != nil {
            if ne, ok := e.(net.Error); ok && ne.Temporary() {
                if tempDelay == 0 {
                    tempDelay = 5 * time.Millisecond
                } else {
                    tempDelay *= 2
                }
                if max := 1 * time.Second; tempDelay > max {
                    tempDelay = max
                }
                log.Printf("http: Accept error: %v; retrying in %v'
                time.Sleep(tempDelay)
                continue
            }
            return e
        }
        tempDelay = 0
        c, err := srv.newConn(rw)
        if err != nil {
            continue
        }
        go c.serve()
    }
}
```

监控之后如何接收客户端的请求呢？上面代码执行监控端口之后，调用了 `srv.Serve(net.Listener)` 函数，这个函数就是处理接收客户端的请求信息。这个函数里面起了一个 `for{}` ，首先通过Listener接收请求，其次创建一个Conn，最后单独开了一个goroutine，把这个请求的数据当做参数扔给这个conn去服务： `go c.serve()` 。这个就是高并发体现了，用户的每一次请求都是在一个新的goroutine去服务，相互不影响。

那么如何具体分配到相应的函数来处理请求呢？conn首先会解析request: `c.readRequest()` ,然后获取相应的handler: `handler := c.server.Handler` ，也就是我们刚才在调用函数 `ListenAndServe` 时候的第二个参数，我们前面例子传递的是nil，也就是为空，那么默认获取 `handler = DefaultServeMux` ,那么这个变量用来做什么的呢？对，这个变量就是一个路由器，它用来匹配url跳转到其相应的handle函数，那么这个我们有设置过吗?有，我们调用的代码里面第一句不是调用了 `http.HandleFunc("/", sayhelloName)` 嘛。这个作用就是注册了请求 `/` 的

路由规则，当请求uri为"/"，路由就会转到函数sayhelloName，DefaultServeMux会调用ServeHTTP方法，这个方法内部其实就是调用sayhelloName本身，最后通过写入response的信息反馈到客户端。

详细的整个流程如下图所示：

图3.10 一个http连接处理流程

至此我们的三个问题已经全部得到了解答，你现在对于Go如何让Web跑起来的是否已经基本了解呢？

links

- 目录
- 上一节: GO搭建一个简单的web服务
- 下一节: Go的http包详解

3.4 Go的http包详解

前面小节介绍了Go怎么样实现了Web工作模式的一个流程，这一小节，我们将详细地解剖一下http包，看它到底是怎样实现整个过程的。

Go的http有两个核心功能：Conn、ServeMux

Conn的goroutine

与我们一般编写的http服务器不同, Go为了实现高并发和高性能, 使用了goroutines来处理Conn的读写事件, 这样每个请求都能保持独立，相互不会阻塞，可以高效的响应网络事件。这是Go高效的保证。

Go在等待客户端请求里面是这样写的：

```
c, err := srv.newConn(rw)
if err != nil {
    continue
}
go c.serve()
```

这里我们可以看到客户端的每次请求都会创建一个Conn，这个Conn里面保存了该次请求的信息，然后再传递到对应的handler，该handler中便可以读取到相应的header信息，这样保证了每个请求的独立性。

ServeMux的自定义

我们前面小节讲述conn.server的时候，其实内部是调用了http包默认的路由器，通过路由器把本次请求的信息传递到了后端的处理函数。那么这个路由器是怎么实现的呢？

它的结构如下：

```
type ServeMux struct {
    mu sync.RWMutex   //锁，由于请求涉及到并发处理，因此这里需要一个锁机制
    m  map[string]muxEntry  // 路由规则，一个string对应一个mux实体，这里
    hosts bool // 是否在任意的规则中带有host信息
}
```

下面看一下muxEntry

```
type muxEntry struct {
    explicit bool   // 是否精确匹配
    h        Handler // 这个路由表达式对应哪个handler
    pattern  string  //匹配字符串
}
```

接着看一下Handler的定义

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)  // 路由实现器
}
```

Handler是一个接口，但是前一小节中的 `sayhelloName` 函数并没有实现ServeHTTP这个接口，为什么能添加呢？原来在http包里面还定义了一个类型 `HandlerFunc` ,我们定义的函数 `sayhelloName` 就是这个HandlerFunc调用之后的结果，这个类型默认就实现了ServeHTTP这个接口，即我们调用了HandlerFunc(f),强制类型转换f成为HandlerFunc类型，这样f就拥有了ServeHTTP方法。

```
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

路由器里面存储好了相应的路由规则之后，那么具体的请求又是怎么分发的呢？请看下面的代码，默认的路由器实现了 `ServeHTTP` ：

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
    if r.RequestURI == "*" {
        w.Header().Set("Connection", "close")
        w.WriteHeader(StatusBadRequest)
        return
    }
    h, _ := mux.Handler(r)
    h.ServeHTTP(w, r)
}
```

如上所示路由器接收到请求之后，如果是 `*` 那么关闭链接，不然调用 `mux.Handler(r)` 返回对应设置路由的处理Handler，然后执行 `h.ServeHTTP(w, r)`

也就是调用对应路由的handler的ServerHTTP接口，那么mux.Handler(r)怎么处理的呢？

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string
    if r.Method != "CONNECT" {
        if p := cleanPath(r.URL.Path); p != r.URL.Path {
            _, pattern = mux.handler(r.Host, p)
            return RedirectHandler(p, StatusMovedPermanently), patt
        }
    }
    return mux.handler(r.Host, r.URL.Path)
}

func (mux *ServeMux) handler(host, path string) (h Handler, pattern
    mux.mu.RLock()
    defer mux.mu.RUnlock()

    // Host-specific pattern takes precedence over generic ones
    if mux.hosts {
        h, pattern = mux.match(host + path)
    }
    if h == nil {
        h, pattern = mux.match(path)
    }
    if h == nil {
        h, pattern = NotFoundHandler(), ""
    }
    return
}
```

原来他是根据用户请求的URL和路由器里面存储的map去匹配的，当匹配到之后返回存储的handler，调用这个handler的ServeHTTP接口就可以执行到相应的函数了。

通过上面这个介绍，我们了解了整个路由过程，Go其实支持外部实现的路由器 `ListenAndServe` 的第二个参数就是用以配置外部路由器的，它是一个Handler接口，即外部路由器只要实现了Handler接口就可以,我们可以在自己实现的路由器的ServeHTTP里面实现自定义路由功能。

如下代码所示，我们自己实现了一个简易的路由器

```
package main

import (
    "fmt"
    "net/http"
)

type MyMux struct {
}

func (p *MyMux) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path == "/" {
        sayhelloName(w, r)
        return
    }
    http.NotFound(w, r)
    return
}

func sayhelloName(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello myroute!")
}

func main() {
    mux := &MyMux{}
    http.ListenAndServe(":9090", mux)
}
```

Go代码的执行流程

通过对http包的分析之后，现在让我们来梳理一下整个的代码执行过程。

- 首先调用Http.HandleFunc

  按顺序做了几件事：

  1 调用了DefaultServeMux的HandleFunc

  2 调用了DefaultServeMux的Handle

  3 往DefaultServeMux的map[string]muxEntry中增加对应的handler和路由规则

- 其次调用http.ListenAndServe(":9090", nil)

  按顺序做了几件事情：

  1 实例化Server

  2 调用Server的ListenAndServe()

3 调用net.Listen("tcp", addr)监听端口

4 启动一个for循环，在循环体中Accept请求

5 对每个请求实例化一个Conn，并且开启一个goroutine为这个请求进行服务go c.serve()

6 读取每个请求的内容w, err := c.readRequest()

7 判断handler是否为空，如果没有设置handler（这个例子就没有设置handler），handler就设置为DefaultServeMux

8 调用handler的ServeHttp

9 在这个例子中，下面就进入到DefaultServeMux.ServeHttp

10 根据request选择handler，并且进入到这个handler的ServeHTTP

```
mux.handler(r).ServeHTTP(w, r)
```

11 选择handler：

A 判断是否有路由能满足这个request（循环遍历ServerMux的muxEntry）

B 如果有路由满足，调用这个路由handler的ServeHttp

C 如果没有路由满足，调用NotFoundHandler的ServeHttp

links

- 目录
- 上一节: Go如何使得web工作
- 下一节: 小结

3.5 小结

这一章我们介绍了HTTP协议, DNS解析的过程, 如何用go实现一个简陋的web server。并深入到net/http包的源码中为大家揭开实现此server的秘密。

希望通过这一章的学习，你能够对Go开发Web有了初步的了解，我们也看到相应的代码了，Go开发Web应用是很方便的，同时又是相当的灵活。

links

- 目录
- 上一节: Go的http包详解
- 下一章: 表单

4 表单

表单是我们平常编写Web应用常用的工具，通过表单我们可以方便的让客户端和服务器进行数据的交互。对于以前开发过Web的用户来说表单都非常熟悉，但是对于C/C++程序员来说，这可能是一个有些陌生的东西，那么什么是表单呢？

表单是一个包含表单元素的区域。表单元素是允许用户在表单中（比如：文本域、下拉列表、单选框、复选框等等）输入信息的元素。表单使用表单标签（\）定义。

```
<form>
...
input 元素
...
</form>
```

Go里面对于form处理已经有很方便的方法了，在Request里面的有专门的form处理，可以很方便的整合到Web开发里面来，4.1小节里面将讲解Go如何处理表单的输入。由于不能信任任何用户的输入，所以我们需要对这些输入进行有效性验证，4.2小节将就如何进行一些普通的验证进行详细的演示。

HTTP协议是一种无状态的协议，那么如何才能辨别是否是同一个用户呢？同时又如何保证一个表单不出现多次递交的情况呢？4.3和4.4小节里面将对cookie(cookie是存储在客户端的信息，能够每次通过header和服务器进行交互的数据)等进行详细讲解。

表单还有一个很大的功能就是能够上传文件，那么Go是如何处理文件上传的呢？针对大文件上传我们如何有效的处理呢？4.5小节我们将一起学习Go处理文件上传的知识。

目录

links

- 目录
- 上一章: 第三章总结
- 下一节: 处理表单的输入

4.1 处理表单的输入

先来看一个表单递交的例子，我们有如下的表单内容，命名成文件login.gtpl(放入当前新建项目的目录里面)

```
<html>
<head>
<title></title>
</head>
<body>
<form action="/login" method="post">
    用户名:<input type="text" name="username">
    密码:<input type="password" name="password">
    <input type="submit" value="登陆">
</form>
</body>
</html>
```

上面递交表单到服务器的 `/login`，当用户输入信息点击登陆之后，会跳转到服务器的路由 `login` 里面，我们首先要判断这个是什么方式传递过来，POST还是GET呢？

http包里面有一个很简单的方式就可以获取，我们在前面web的例子的基础上来看看怎么处理login页面的form数据

```
package main

import (
    "fmt"
    "html/template"
    "log"
    "net/http"
    "strings"
)

func sayhelloName(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()       //解析url传递的参数，对于POST则解析响应包的主体（
    //注意:如果没有调用ParseForm方法，下面无法获取表单的数据
    fmt.Println(r.Form) //这些信息是输出到服务器端的打印信息
    fmt.Println("path", r.URL.Path)
    fmt.Println("scheme", r.URL.Scheme)
    fmt.Println(r.Form["url_long"])
    for k, v := range r.Form {
        fmt.Println("key:", k)
        fmt.Println("val:", strings.Join(v, ""))
    }
    fmt.Fprintf(w, "Hello astaxie!") //这个写入到w的是输出到客户端的
}

func login(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method:", r.Method) //获取请求的方法
    if r.Method == "GET" {
        t, _ := template.ParseFiles("login.gtpl")
        t.Execute(w, nil)
    } else {
        //请求的是登陆数据，那么执行登陆的逻辑判断
        fmt.Println("username:", r.Form["username"])
        fmt.Println("password:", r.Form["password"])
    }
}

func main() {
    http.HandleFunc("/", sayhelloName)       //设置访问的路由
    http.HandleFunc("/login", login)         //设置访问的路由
    err := http.ListenAndServe(":9090", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

通过上面的代码我们可以看出获取请求方法是通过 `r.Method` 来完成的，这是个字符串类型的变量，返回GET, POST, PUT等method信息。

login函数中我们根据 `r.Method` 来判断是显示登录界面还是处理登录逻辑。当GET方式请求时显示登录界面，其他方式请求时则处理登录逻辑，如查询数据库、验证登录信息等。

当我们在浏览器里面打开 `http://127.0.0.1:9090/login` 的时候，出现如下界面

图4.1 用户登录界面

我们输入用户名和密码之后发现在服务器端是不会打印出来任何输出的，为什么呢？默认情况下，Handler里面是不会自动解析form的，必须显式的调用 `r.ParseForm()` 后，你才能对这个表单数据进行操作。我们修改一下代码，在 `fmt.Println("username:", r.Form["username"])` 之前加一行 `r.ParseForm()` ,重新编译，再次测试输入递交，现在是不是在服务器端有输出你的输入的用户名和密码了。

`r.Form` 里面包含了所有请求的参数，比如URL中query-string、POST的数据、PUT的数据，所有当你在URL的query-string字段和POST冲突时，会保存成一个slice，里面存储了多个值，Go官方文档中说在接下来的版本里面将会把POST、GET这些数据分离开来。

现在我们修改一下login.gtpl里面form的action值 `http://127.0.0.1:9090/login` 修改为 `http://127.0.0.1:9090/login?username=astaxie` ，再次测试，服务器的输出username是不是一个slice。服务器端的输出如下：

图4.2 服务器端打印接受到的信息

`request.Form` 是一个url.Values类型，里面存储的是对应的类似 `key=value` 的信息，下面展示了可以对form数据进行的一些操作:

```
v := url.Values{}
v.Set("name", "Ava")
v.Add("friend", "Jess")
v.Add("friend", "Sarah")
v.Add("friend", "Zoe")
// v.Encode() == "name=Ava&friend=Jess&friend=Sarah&friend=Zoe"
fmt.Println(v.Get("name"))
fmt.Println(v.Get("friend"))
fmt.Println(v["friend"])
```

> **Tips**: Request本身也提供了FormValue()函数来获取用户提交的参数。如r.Form["username"]也可写成r.FormValue("username")。调用r.FormValue时会自动调用r.ParseForm，所以不必提前调用。r.FormValue只会返回同名参数中的第一个，若参数不存在则返回空字符串。

links

- 目录
- 上一节: 表单
- 下一节: 验证表单的输入

4.2 验证表单的输入

开发Web的一个原则就是，不能信任用户输入的任何信息，所以验证和过滤用户的输入信息就变得非常重要，我们经常会在微博、新闻中听到某某网站被入侵了，存在什么漏洞，这些大多是因为网站对于用户输入的信息没有做严格的验证引起的，所以为了编写出安全可靠的Web程序，验证表单输入的意义重大。

我们平常编写Web应用主要有两方面的数据验证，一个是在页面端的js验证(目前在这方面有很多的插件库，比如ValidationJS插件)，一个是在服务器端的验证，我们这小节讲解的是如何在服务器端验证。

必填字段

你想要确保从一个表单元素中得到一个值，例如前面小节里面的用户名，我们如何处理呢？Go有一个内置函数 `len` 可以获取字符串的长度，这样我们就可以通过len来获取数据的长度，例如：

```
if len(r.Form["username"][0])==0{
    //为空的处理
}
```

`r.Form` 对不同类型的表单元素的留空有不同的处理， 对于空文本框、空文本区域以及文件上传，元素的值为空值,而如果是未选中的复选框和单选按钮，则根本不会在r.Form中产生相应条目，如果我们用上面例子中的方式去获取数据时程序就会报错。所以我们需要通过 `r.Form.Get()` 来获取值，因为如果字段不存在，通过该方式获取的是空值。但是通过 `r.Form.Get()` 只能获取单个的值，如果是map的值，必须通过上面的方式来获取。

数字

你想要确保一个表单输入框中获取的只能是数字，例如，你想通过表单获取某个人的具体年龄是50岁还是10岁，而不是像“一把年纪了”或“年轻着呢”这种描述

如果我们是判断正整数，那么我们先转化成int类型，然后进行处理

```
getint,err:=strconv.Atoi(r.Form.Get("age"))
if err!=nil{
    //数字转化出错了，那么可能就不是数字
}

//接下来就可以判断这个数字的大小范围了
if getint >100 {
    //太大了
}
```

还有一种方式就是正则匹配的方式

```
if m, _ := regexp.MatchString("^[0-9]+$", r.Form.Get("age")); !m {
    return false
}
```

对于性能要求很高的用户来说，这是一个老生常谈的问题了，他们认为应该尽量避免使用正则表达式，因为使用正则表达式的速度会比较慢。但是在目前机器性能那么强劲的情况下，对于这种简单的正则表达式效率和类型转换函数是没有什么差别的。如果你对正则表达式很熟悉，而且你在其它语言中也在使用它，那么在Go里面使用正则表达式将是一个便利的方式。

> Go实现的正则是RE2，所有的字符都是UTF-8编码的。

中文

有时候我们想通过表单元素获取一个用户的中文名字，但是又为了保证获取的是正确的中文，我们需要进行验证，而不是用户随便的一些输入。对于中文我们目前有两种方式来验证，可以使用 `unicode` 包提供的 `func Is(rangeTab *RangeTable, r rune) bool` 来验证，也可以使用正则方式来验证，这里使用最简单的正则方式，如下代码所示

```
if m, _ := regexp.MatchString("^\\p{Han}+$", r.Form.Get("realname")
    return false
}
```

英文

我们期望通过表单元素获取一个英文值，例如我们想知道一个用户的英文名，应该是astaxie，而不是asta谢。

我们可以很简单的通过正则验证数据：

```
if m, _ := regexp.MatchString("^[a-zA-Z]+$", r.Form.Get("engname"))
    return false
}
```

电子邮件地址

你想知道用户输入的一个Email地址是否正确，通过如下这个方式可以验证：

```
if m, _ := regexp.MatchString(`^([\w\.\_]{2,10})@(\w{1,}).([a-z]{2,
    fmt.Println("no")
}else{
    fmt.Println("yes")
}
```

手机号码

你想要判断用户输入的手机号码是否正确，通过正则也可以验证：

```
if m, _ := regexp.MatchString(`^(1[3|4|5|8][0-9]\d{4,8})$`, r.Form
    return false
}
```

下拉菜单

如果我们想要判断表单里面 `<select>` 元素生成的下拉菜单中是否有被选中的项目。有些时候黑客可能会伪造这个下拉菜单不存在的值发送给你，那么如何判断这个值是否是我们预设的值呢？

我们的select可能是这样的一些元素

```
<select name="fruit">
<option value="apple">apple</option>
<option value="pear">pear</option>
<option value="banane">banane</option>
</select>
```

那么我们可以这样来验证

```
slice:=[]string{"apple","pear","banane"}

for _, v := range slice {
    if v == r.Form.Get("fruit") {
        return true
    }
}
return false
```

单选按钮

如果我们想要判断radio按钮是否有一个被选中了，我们页面的输出可能就是一个男、女性别的选择，但是也可能一个15岁大的无聊小孩，一手拿着http协议的书，另一只手通过telnet客户端向你的程序在发送请求呢，你设定的性别男值是1，女是2，他给你发送一个3，你的程序会出现异常吗？因此我们也需要像下拉菜单的判断方式类似，判断我们获取的值是我们预设的值，而不是额外的值。

```
<input type="radio" name="gender" value="1">男
<input type="radio" name="gender" value="2">女
```

那我们也可以类似下拉菜单的做法一样

```
slice:=[]int{1,2}

for _, v := range slice {
    if v == r.Form.Get("gender") {
        return true
    }
}
return false
```

复选框

有一项选择兴趣的复选框，你想确定用户选中的和你提供给用户选择的是同一个类型的数据。

```
<input type="checkbox" name="interest" value="football">足球
<input type="checkbox" name="interest" value="basketball">篮球
<input type="checkbox" name="interest" value="tennis">网球
```

对于复选框我们的验证和单选有点不一样，因为接收到的数据是一个slice

```
slice:=[]string{"football","basketball","tennis"}
a:=Slice_diff(r.Form["interest"],slice)
if a == nil{
    return true
}

return false
```

上面这个函数 `Slice_diff` 包含在我开源的一个库里面(操作slice和map的库)，https://github.com/astaxie/beeku

日期和时间

你想确定用户填写的日期或时间是否有效。例如，用户在日程表中安排8月份的第45天开会，或者提供未来的某个时间作为生日。

Go里面提供了一个time的处理包，我们可以把用户的输入年月日转化成相应的时间，然后进行逻辑判断

```
t := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
fmt.Printf("Go launched at %s\n", t.Local())
```

获取time之后我们就可以进行很多时间函数的操作。具体的判断就根据自己的需求调整。

身份证号码

如果我们想验证表单输入的是否是身份证，通过正则也可以方便的验证，但是身份证有15位和18位，我们两个都需要验证

```
//验证15位身份证，15位的是全部数字
if m, _ := regexp.MatchString(`^(\d{15})$`, r.Form.Get("usercard")
    return false
}

//验证18位身份证，18位前17位为数字，最后一位是校验位，可能为数字或字符X。
if m, _ := regexp.MatchString(`^(\d{17})([0-9]|X)$`, r.Form.Get("us
    return false
}
```

上面列出了我们一些常用的服务器端的表单元素验证，希望通过这个引导入门，能够让你对Go的数据验证有所了解，特别是Go里面的正则处理。

links

- 目录
- 上一节: 处理表单的输入
- 下一节: 预防跨站脚本

4.3 预防跨站脚本

现在的网站包含大量的动态内容以提高用户体验，比过去要复杂得多。所谓动态内容，就是根据用户环境和需要，Web应用程序能够输出相应的内容。动态站点会受到一种名为“跨站脚本攻击”（Cross Site Scripting, 安全专家们通常将其缩写成XSS）的威胁，而静态站点则完全不受其影响。

攻击者通常会在有漏洞的程序中插入JavaScript、VBScript、 ActiveX或Flash以欺骗用户。一旦得手，他们可以盗取用户帐户信息，修改用户设置，盗取/污染cookie和植入恶意广告等。

对XSS最佳的防护应该结合以下两种方法：一是验证所有输入数据，有效检测攻击(这个我们前面小节已经有过介绍);另一个是对所有输出数据进行适当的处理，以防止任何已成功注入的脚本在浏览器端运行。

那么Go里面是怎么做这个有效防护的呢？Go的html/template里面带有下面几个函数可以帮你转义

- func HTMLEscape(w io.Writer, b []byte) //把b进行转义之后写到w
- func HTMLEscapeString(s string) string //转义s之后返回结果字符串
- func HTMLEscaper(args ...interface{}) string //支持多个参数一起转义，返回结果字符串

我们看4.1小节的例子

```
fmt.Println("username:", template.HTMLEscapeString(r.Form.Get("user
fmt.Println("password:", template.HTMLEscapeString(r.Form.Get("pass
template.HTMLEscape(w, []byte(r.Form.Get("username"))) //输出到客户端
```

如果我们输入的username是 `<script>alert()</script>` ,那么我们可以在浏览器上面看到输出如下所示：

图4.3 Javascript过滤之后的输出

Go的html/template包默认帮你过滤了html标签，但是有时候你只想要输出这个 `<script>alert()</script>` 看起来正常的信息，该怎么处理？请使用text/template。请看下面的例子：

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{e
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwn
```

输出

```
Hello, <script>alert('you have been pwned')</script>!
```

或者使用template.HTML类型

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{e
err = t.ExecuteTemplate(out, "T", template.HTML("<script>alert('you
```

输出

```
Hello, <script>alert('you have been pwned')</script>!
```

转换成 `template.HTML` 后，变量的内容也不会被转义

转义的例子：

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{e
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwn
```

转义之后的输出：

```
Hello, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/scrip
```

links

- 目录
- 上一节: 验证的输入
- 下一节: 防止多次递交表单

4.4 防止多次递交表单

不知道你是否曾经看到过一个论坛或者博客，在一个帖子或者文章后面出现多条重复的记录，这些大多数是因为用户重复递交了留言的表单引起的。由于种种原因，用户经常会重复递交表单。通常这只是鼠标的误操作，如双击了递交按钮，也可能是为了编辑或者再次核对填写过的信息，点击了浏览器的后退按钮，然后又再次点击了递交按钮而不是浏览器的前进按钮。当然，也可能是故意的——比如，在某项在线调查或者博彩活动中重复投票。那我们如何有效的防止用户多次递交相同的表单呢？

解决方案是在表单中添加一个带有唯一值的隐藏字段。在验证表单时，先检查带有该惟一值的表单是否已经递交过了。如果是，拒绝再次递交；如果不是，则处理表单进行逻辑处理。另外，如果是采用了Ajax模式递交表单的话，当表单递交后，通过javascript来禁用表单的递交按钮。

我继续拿4.2小节的例子优化：

```
<input type="checkbox" name="interest" value="football">足球
<input type="checkbox" name="interest" value="basketball">篮球
<input type="checkbox" name="interest" value="tennis">网球
用户名:<input type="text" name="username">
密码:<input type="password" name="password">
<input type="hidden" name="token" value="{{.}}">
<input type="submit" value="登陆">
```

我们在模版里面增加了一个隐藏字段 `token` ，这个值我们通过MD5(时间戳)来获取惟一值，然后我们把这个值存储到服务器端(session来控制，我们将在第六章讲解如何保存)，以方便表单提交时比对判定。

```
func login(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method:", r.Method) //获取请求的方法
    if r.Method == "GET" {
        crutime := time.Now().Unix()
        h := md5.New()
        io.WriteString(h, strconv.FormatInt(crutime, 10))
        token := fmt.Sprintf("%x", h.Sum(nil))

        t, _ := template.ParseFiles("login.gtpl")
        t.Execute(w, token)
    } else {
        //请求的是登陆数据，那么执行登陆的逻辑判断
        r.ParseForm()
        token := r.Form.Get("token")
        if token != "" {
            //验证token的合法性
        } else {
            //不存在token报错
        }
        fmt.Println("username length:", len(r.Form["username"][0]))
        fmt.Println("username:", template.HTMLEscapeString(r.Form.
        fmt.Println("password:", template.HTMLEscapeString(r.Form.
        template.HTMLEscape(w, []byte(r.Form.Get("username"))) //输
    }
}
```

上面的代码输出到页面的源码如下：

图4.4 增加token之后在客户端输出的源码信息

我们看到token已经有输出值，你可以不断的刷新，可以看到这个值在不断的变化。这样就保证了每次显示form表单的时候都是唯一的，用户递交的表单保持了唯一性。

我们的解决方案可以防止非恶意的攻击，并能使恶意用户暂时不知所措，然后，它却不能排除所有的欺骗性的动机，对此类情况还需要更复杂的工作。

links

- 目录
- 上一节: 预防跨站脚本
- 下一节: 处理文件上传

4.5 处理文件上传

你想处理一个由用户上传的文件，比如你正在建设一个类似Instagram的网站，你需要存储用户拍摄的照片。这种需求该如何实现呢？

要使表单能够上传文件，首先第一步就是要添加form的 `enctype` 属性， `enctype` 属性有如下三种情况:

```
application/x-www-form-urlencoded   表示在发送前编码所有字符（默认）
multipart/form-data      不对字符编码。在使用包含文件上传控件的表单时，必须
text/plain      空格转换为 "+" 加号，但不对特殊字符编码。
```

所以，表单的html代码应该类似于:

```
<html>
<head>
    <title>上传文件</title>
</head>
<body>
<form enctype="multipart/form-data" action="http://127.0.0.1:9090/u
  <input type="file" name="uploadfile" />
  <input type="hidden" name="token" value="{{.}}"/>
  <input type="submit" value="upload" />
</form>
</body>
</html>
```

在服务器端，我们增加一个handlerFunc:

```
http.HandleFunc("/upload", upload)

// 处理/upload 逻辑
func upload(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method:", r.Method) //获取请求的方法
    if r.Method == "GET" {
        crutime := time.Now().Unix()
        h := md5.New()
        io.WriteString(h, strconv.FormatInt(crutime, 10))
        token := fmt.Sprintf("%x", h.Sum(nil))

        t, _ := template.ParseFiles("upload.gtpl")
        t.Execute(w, token)
    } else {
        r.ParseMultipartForm(32 << 20)
        file, handler, err := r.FormFile("uploadfile")
        if err != nil {
            fmt.Println(err)
            return
        }
        defer file.Close()
        fmt.Fprintf(w, "%v", handler.Header)
        f, err := os.OpenFile("./test/"+handler.Filename, os.O_WRON
        if err != nil {
            fmt.Println(err)
            return
        }
        defer f.Close()
        io.Copy(f, file)
    }
}
```

通过上面的代码可以看到，处理文件上传我们需要调用 `r.ParseMultipartForm` ，里面的参数表示 `maxMemory` ，调用 `ParseMultipartForm` 之后，上传的文件存储在 `maxMemory` 大小的内存里面，如果文件大小超过了 `maxMemory` ，那么剩下的部分将存储在系统的临时文件中。我们可以通过 `r.FormFile` 获取上面的文件句柄，然后实例中使用了 `io.Copy` 来存储文件。

> 获取其他非文件字段信息的时候就不需要调用 `r.ParseForm` ，因为在需要的时候Go自动会去调用。而且 `ParseMultipartForm` 调用一次之后，后面再次调用不会再有效果。

通过上面的实例我们可以看到我们上传文件主要三步处理：

1. 表单中增加enctype="multipart/form-data"
2. 服务端调用 `r.ParseMultipartForm` ,把上传的文件存储在内存和临时文件中
3. 使用 `r.FormFile` 获取文件句柄，然后对文件进行存储等处理。

文件handler是multipart.FileHeader,里面存储了如下结构信息

```
type FileHeader struct {
    Filename string
    Header   textproto.MIMEHeader
    // contains filtered or unexported fields
}
```

我们通过上面的实例代码打印出来上传文件的信息如下

图4.5 打印文件上传后服务器端接受的信息

客户端上传文件

我们上面的例子演示了如何通过表单上传文件，然后在服务器端处理文件，其实Go支持模拟客户端表单功能支持文件上传，详细用法请看如下示例：

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "io/ioutil"
    "mime/multipart"
    "net/http"
    "os"
)

func postFile(filename string, targetUrl string) error {
    bodyBuf := &bytes.Buffer{}
    bodyWriter := multipart.NewWriter(bodyBuf)

    //关键的一步操作
    fileWriter, err := bodyWriter.CreateFormFile("uploadfile", file
    if err != nil {
        fmt.Println("error writing to buffer")
        return err
    }

    //打开文件句柄操作
    fh, err := os.Open(filename)
    if err != nil {
        fmt.Println("error opening file")
        return err
    }
    defer fh.Close()
```

```
    //iocopy
    _, err = io.Copy(fileWriter, fh)
    if err != nil {
        return err
    }

    contentType := bodyWriter.FormDataContentType()
    bodyWriter.Close()

    resp, err := http.Post(targetUrl, contentType, bodyBuf)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    resp_body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return err
    }
    fmt.Println(resp.Status)
    fmt.Println(string(resp_body))
    return nil
}

// sample usage
func main() {
    target_url := "http://localhost:9090/upload"
    filename := "./astaxie.pdf"
    postFile(filename, target_url)
}
```

上面的例子详细展示了客户端如何向服务器上传一个文件的例子，客户端通过multipart.Write把文件的文本流写入一个缓存中，然后调用http的Post方法把缓存传到服务器。

> 如果你还有其他普通字段例如username之类的需要同时写入，那么可以调用multipart的WriteField方法写很多其他类似的字段。

links

- 目录
- 上一节: 防止多次递交表单
- 下一节: 小结

4.6 小结

这一章里面我们学习了Go如何处理表单信息，我们通过用户登陆、上传文件的例子展示了Go处理form表单信息及上传文件的手段。但是在处理表单过程中我们需要验证用户输入的信息，考虑到网站安全的重要性，数据过滤就显得相当重要了，因此后面的章节中专门写了一个小节来讲解了不同方面的数据过滤，顺带讲一下Go对字符串的正则处理。

通过这一章能够让你了解客户端和服务器端是如何进行数据上的交互，客户端将数据传递给服务器系统，服务器接受数据又把处理结果反馈给客户端。

links

- 目录
- 上一节: 处理文件上传
- 下一章: 访问数据库

5 访问数据库

对许多Web应用程序而言，数据库都是其核心所在。数据库几乎可以用来存储你想查询和修改的任何信息，比如用户信息、产品目录或者新闻列表等。

Go没有内置的驱动支持任何的数据库，但是Go定义了database/sql接口，用户可以基于驱动接口开发相应数据库的驱动，5.1小节里面介绍Go设计的一些驱动，介绍Go是如何设计数据库驱动接口的。5.2至5.4小节介绍目前使用的比较多的一些关系型数据驱动以及如何使用，5.5小节介绍我自己开发一个ORM库，基于database/sql标准接口开发的，可以兼容几乎所有支持database/sql的数据库驱动，可以方便的使用Go style来进行数据库操作。

目前NOSQL已经成为Web开发的一个潮流，很多应用采用了NOSQL作为数据库，而不是以前的缓存，5.6小节将介绍MongoDB和Redis两种NOSQL数据库。

> Go database/sql tutorial 里提供了惯用的范例及详细的说明。

目录

links

- 目录
- 上一章: 第四章总结
- 下一节: database/sql接口

5.1 database/sql接口

Go与PHP不同的地方是Go官方没有提供数据库驱动，而是为开发数据库驱动定义了一些标准接口，开发者可以根据定义的接口来开发相应的数据库驱动，这样做有一个好处，只要是按照标准接口开发的代码， 以后需要迁移数据库时，不需要任何修改。那么Go都定义了哪些标准接口呢？让我们来详细的分析一下

sql.Register

这个存在于database/sql的函数是用来注册数据库驱动的，当第三方开发者开发数据库驱动时，都会实现init函数，在init里面会调用这个 `Register(name string, driver driver.Driver)` 完成本驱动的注册。

我们来看一下mymysql、sqlite3的驱动里面都是怎么调用的：

```
//https://github.com/mattn/go-sqlite3驱动
func init() {
    sql.Register("sqlite3", &SQLiteDriver{})
}

//https://github.com/mikespook/mymysql驱动
// Driver automatically registered in database/sql
var d = Driver{proto: "tcp", raddr: "127.0.0.1:3306"}
func init() {
    Register("SET NAMES utf8")
    sql.Register("mymysql", &d)
}
```

我们看到第三方数据库驱动都是通过调用这个函数来注册自己的数据库驱动名称以及相应的driver实现。在database/sql内部通过一个map来存储用户定义的相应驱动。

```
var drivers = make(map[string]driver.Driver)

drivers[name] = driver
```

因此通过database/sql的注册函数可以同时注册多个数据库驱动，只要不重复。

> 在我们使用database/sql接口和第三方库的时候经常看到如下:
>
> ```
> import (
>     "database/sql"
>      _ "github.com/mattn/go-sqlite3"
> )
> ```
>
> 新手都会被这个 `_` 所迷惑，其实这个就是Go设计的巧妙之处，我们在变量赋值的时候经常看到这个符号，它是用来忽略变量赋值的占位符，那么包引入用到这个符号也是相似的作用，这儿使用 `_` 的意思是引入后面的包名而不直接使用这个包中定义的函数，变量等资源。
>
> 我们在2.3节流程和函数一节中介绍过init函数的初始化过程，包在引入的时候会自动调用包的init函数以完成对包的初始化。因此，我们引入上面的数据库驱动包之后会自动去调用init函数，然后在init函数里面注册这个数据库驱动，这样我们就可以在接下来的代码中直接使用这个数据库驱动了。

driver.Driver

Driver是一个数据库驱动的接口，他定义了一个method： Open(name string)，这个方法返回一个数据库的Conn接口。

```
type Driver interface {
    Open(name string) (Conn, error)
}
```

返回的Conn只能用来进行一次goroutine的操作，也就是说不能把这个Conn应用于Go的多个goroutine里面。如下代码会出现错误

```
...
go goroutineA (Conn)  //执行查询操作
go goroutineB (Conn)  //执行插入操作
...
```

上面这样的代码可能会使Go不知道某个操作究竟是由哪个goroutine发起的,从而导致数据混乱，比如可能会把goroutineA里面执行的查询操作的结果返回给goroutineB从而使B错误地把此结果当成自己执行的插入数据。

第三方驱动都会定义这个函数，它会解析name参数来获取相关数据库的连接信息，解析完成后，它将使用此信息来初始化一个Conn并返回它。

driver.Conn

Conn是一个数据库连接的接口定义，他定义了一系列方法，这个Conn只能应用在一个goroutine里面，不能使用在多个goroutine里面，详情请参考上面的说明。

```
type Conn interface {
    Prepare(query string) (Stmt, error)
    Close() error
    Begin() (Tx, error)
}
```

Prepare函数返回与当前连接相关的执行Sql语句的准备状态，可以进行查询、删除等操作。

Close函数关闭当前的连接，执行释放连接拥有的资源等清理工作。因为驱动实现了database/sql里面建议的conn pool，所以你不用再去实现缓存conn之类的，这样会容易引起问题。

Begin函数返回一个代表事务处理的Tx，通过它你可以进行查询,更新等操作，或者对事务进行回滚、递交。

driver.Stmt

Stmt是一种准备好的状态，和Conn相关联，而且只能应用于一个goroutine中，不能应用于多个goroutine。

```
type Stmt interface {
    Close() error
    NumInput() int
    Exec(args []Value) (Result, error)
    Query(args []Value) (Rows, error)
}
```

Close函数关闭当前的链接状态，但是如果当前正在执行query，query还是有效返回rows数据。

NumInput函数返回当前预留参数的个数，当返回>=0时数据库驱动就会智能检查调用者的参数。当数据库驱动包不知道预留参数的时候，返回-1。

Exec函数执行Prepare准备好的sql，传入参数执行update/insert等操作，返回Result数据

Query函数执行Prepare准备好的sql，传入需要的参数执行select操作，返回Rows结果集

driver.Tx

事务处理一般就两个过程，递交或者回滚。数据库驱动里面也只需要实现这两个函数就可以

```
type Tx interface {
    Commit() error
    Rollback() error
}
```

这两个函数一个用来递交一个事务，一个用来回滚事务。

driver.Execer

这是一个Conn可选择实现的接口

```
type Execer interface {
    Exec(query string, args []Value) (Result, error)
}
```

如果这个接口没有定义，那么在调用DB.Exec,就会首先调用Prepare返回Stmt，然后执行Stmt的Exec，然后关闭Stmt。

driver.Result

这个是执行Update/Insert等操作返回的结果接口定义

```
type Result interface {
    LastInsertId() (int64, error)
    RowsAffected() (int64, error)
}
```

LastInsertId函数返回由数据库执行插入操作得到的自增ID号。

RowsAffected函数返回query操作影响的数据条目数。

driver.Rows

Rows是执行查询返回的结果集接口定义

```
type Rows interface {
    Columns() []string
    Close() error
    Next(dest []Value) error
}
```

Columns函数返回查询数据库表的字段信息，这个返回的slice和sql查询的字段一一对应，而不是返回整个表的所有字段。

Close函数用来关闭Rows迭代器。

Next函数用来返回下一条数据，把数据赋值给dest。dest里面的元素必须是driver.Value的值除了string，返回的数据里面所有的string都必须要转换成[]byte。如果最后没数据了，Next函数最后返回io.EOF。

driver.RowsAffected

RowsAffected其实就是一个int64的别名，但是他实现了Result接口，用来底层实现Result的表示方式

```
type RowsAffected int64

func (RowsAffected) LastInsertId() (int64, error)

func (v RowsAffected) RowsAffected() (int64, error)
```

driver.Value

Value其实就是一个空接口，他可以容纳任何的数据

```
type Value interface{}
```

drive的Value是驱动必须能够操作的Value，Value要么是nil，要么是下面的任意一种

```
int64
float64
bool
[]byte
string   [*]除了Rows.Next返回的不能是string.
time.Time
```

driver.ValueConverter

ValueConverter接口定义了如何把一个普通的值转化成driver.Value的接口

```
type ValueConverter interface {
    ConvertValue(v interface{}) (Value, error)
}
```

在开发的数据库驱动包里面实现这个接口的函数在很多地方会使用到，这个ValueConverter有很多好处：

- 转化driver.value到数据库表相应的字段，例如int64的数据如何转化成数据库表uint16字段
- 把数据库查询结果转化成driver.Value值
- 在scan函数里面如何把driver.Value值转化成用户定义的值

driver.Valuer

Valuer接口定义了返回一个driver.Value的方式

```
type Valuer interface {
    Value() (Value, error)
}
```

很多类型都实现了这个Value方法，用来自身与driver.Value的转化。

通过上面的讲解，你应该对于驱动的开发有了一个基本的了解，一个驱动只要实现了这些接口就能完成增删查改等基本操作了，剩下的就是与相应的数据库进行数据交互等细节问题了，在此不再赘述。

database/sql

database/sql在database/sql/driver提供的接口基础上定义了一些更高阶的方法，用以简化数据库操作,同时内部还建议性地实现一个conn pool。

```
type DB struct {
    driver      driver.Driver
    dsn         string
    mu       sync.Mutex // protects freeConn and closed
    freeConn []driver.Conn
    closed   bool
}
```

我们可以看到Open函数返回的是DB对象，里面有一个freeConn，它就是那个简易的连接池。它的实现相当简单或者说简陋，就是当执行Db.prepare的时候会`defer db.putConn(ci, err)`,也就是把这个连接放入连接池，每次调用conn的时候会先判断freeConn的长度是否大于0，大于0说明有可以复用的conn，直接拿出来用就是了，如果不大于0，则创建一个conn,然后再返回之。

links

- 目录
- 上一节: 访问数据库
- 下一节: 使用MySQL数据库

5.2 使用MySQL数据库

目前Internet上流行的网站构架方式是LAMP，其中的M即MySQL, 作为数据库，MySQL以免费、开源、使用方便为优势成为了很多Web开发的后端数据库存储引擎。

MySQL驱动

Go中支持MySQL的驱动目前比较多，有如下几种，有些是支持database/sql标准，而有些是采用了自己的实现接口,常用的有如下几种:

- https://github.com/go-sql-driver/mysql 支持database/sql，全部采用go写。
- https://github.com/ziutek/mymysql 支持database/sql，也支持自定义的接口，全部采用go写。
- https://github.com/Philio/GoMySQL 不支持database/sql，自定义接口，全部采用go写。

接下来的例子我主要以第一个驱动为例(我目前项目中也是采用它来驱动)，也推荐大家采用它，主要理由：

- 这个驱动比较新，维护的比较好
- 完全支持database/sql接口
- 支持keepalive，保持长连接,虽然星星fork的mymysql也支持keepalive，但不是线程安全的，这个从底层就支持了keepalive。

示例代码

接下来的几个小节里面我们都将采用同一个数据库表结构：数据库test，用户表userinfo，关联用户信息表userdetail。

```
CREATE TABLE `userinfo` (
    `uid` INT(10) NOT NULL AUTO_INCREMENT,
    `username` VARCHAR(64) NULL DEFAULT NULL,
    `departname` VARCHAR(64) NULL DEFAULT NULL,
    `created` DATE NULL DEFAULT NULL,
    PRIMARY KEY (`uid`)
)

CREATE TABLE `userdetail` (
    `uid` INT(10) NOT NULL DEFAULT '0',
    `intro` TEXT NULL,
    `profile` TEXT NULL,
    PRIMARY KEY (`uid`)
)
```

如下示例将示范如何使用database/sql接口对数据库表进行增删改查操作

```
package main

import (
    _ "github.com/go-sql-driver/mysql"
    "database/sql"
    "fmt"
    //"time"
)

func main() {
    db, err := sql.Open("mysql", "astaxie:astaxie@/test?charset=utf
    checkErr(err)

    //插入数据
    stmt, err := db.Prepare("INSERT userinfo SET username=?,departm
    checkErr(err)

    res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")
    checkErr(err)

    id, err := res.LastInsertId()
    checkErr(err)

    fmt.Println(id)
    //更新数据
    stmt, err = db.Prepare("update userinfo set username=? where ui
    checkErr(err)

    res, err = stmt.Exec("astaxieupdate", id)
    checkErr(err)

    affect, err := res.RowsAffected()
    checkErr(err)

    fmt.Println(affect)

    //查询数据
    rows, err := db.Query("SELECT * FROM userinfo")
    checkErr(err)

    for rows.Next() {
        var uid int
        var username string
        var department string
        var created string
        err = rows.Scan(&uid, &username, &department, &created)
        checkErr(err)
        fmt.Println(uid)
        fmt.Println(username)
        fmt.Println(department)
```

```
            fmt.Println(created)
        }

        //删除数据
        stmt, err = db.Prepare("delete from userinfo where uid=?")
        checkErr(err)

        res, err = stmt.Exec(id)
        checkErr(err)

        affect, err = res.RowsAffected()
        checkErr(err)

        fmt.Println(affect)

        db.Close()

    }

    func checkErr(err error) {
        if err != nil {
            panic(err)
        }
    }
```

通过上面的代码我们可以看出，Go操作Mysql数据库是很方便的。

关键的几个函数我解释一下：

sql.Open()函数用来打开一个注册过的数据库驱动，go-sql-driver中注册了mysql这个数据库驱动，第二个参数是DSN(Data Source Name)，它是go-sql-driver定义的一些数据库链接和配置信息。它支持如下格式：

```
user@unix(/path/to/socket)/dbname?charset=utf8
user:password@tcp(localhost:5555)/dbname?charset=utf8
user:password@/dbname
user:password@tcp([de:ad:be:ef::ca:fe]:80)/dbname
```

db.Prepare()函数用来返回准备要执行的sql操作，然后返回准备完毕的执行状态。

db.Query()函数用来直接执行Sql返回Rows结果。

stmt.Exec()函数用来执行stmt准备好的SQL语句

我们可以看到我们传入的参数都是=?对应的数据，这样做的方式可以一定程度上防止SQL注入。

links

- 目录
- 上一节: database/sql接口
- 下一节: 使用SQLite数据库

5.3 使用SQLite数据库

SQLite 是一个开源的嵌入式关系数据库，实现自包容、零配置、支持事务的SQL数据库引擎。其特点是高度便携、使用方便、结构紧凑、高效、可靠。 与其他数据库管理系统不同，SQLite 的安装和运行非常简单，在大多数情况下,只要确保SQLite的二进制文件存在即可开始创建、连接和使用数据库。如果您正在寻找一个嵌入式数据库项目或解决方案，SQLite是绝对值得考虑。SQLite可以是说开源的Access。

驱动

Go支持sqlite的驱动也比较多，但是好多都是不支持database/sql接口的

- https://github.com/mattn/go-sqlite3 支持database/sql接口，基于cgo(关于cgo的知识请参看官方文档或者本书后面的章节)写的
- https://github.com/feyeleanor/gosqlite3 不支持database/sql接口，基于cgo写的
- https://github.com/phf/go-sqlite3 不支持database/sql接口，基于cgo写的

目前支持database/sql的SQLite数据库驱动只有第一个，我目前也是采用它来开发项目的。采用标准接口有利于以后出现更好的驱动的时候做迁移。

实例代码

示例的数据库表结构如下所示，相应的建表SQL：

```
CREATE TABLE `userinfo` (
    `uid` INTEGER PRIMARY KEY AUTOINCREMENT,
    `username` VARCHAR(64) NULL,
    `departname` VARCHAR(64) NULL,
    `created` DATE NULL
);

CREATE TABLE `userdeatail` (
    `uid` INT(10) NULL,
    `intro` TEXT NULL,
    `profile` TEXT NULL,
    PRIMARY KEY (`uid`)
);
```

看下面Go程序是如何操作数据库表数据:增删改查

```
package main

import (
```

```
    "database/sql"
    "fmt"
    "time"
    _ "github.com/mattn/go-sqlite3"
)

func main() {
    db, err := sql.Open("sqlite3", "./foo.db")
    checkErr(err)

    //插入数据
    stmt, err := db.Prepare("INSERT INTO userinfo(username, departr
    checkErr(err)

    res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")
    checkErr(err)

    id, err := res.LastInsertId()
    checkErr(err)

    fmt.Println(id)
    //更新数据
    stmt, err = db.Prepare("update userinfo set username=? where ui
    checkErr(err)

    res, err = stmt.Exec("astaxieupdate", id)
    checkErr(err)

    affect, err := res.RowsAffected()
    checkErr(err)

    fmt.Println(affect)

    //查询数据
    rows, err := db.Query("SELECT * FROM userinfo")
    checkErr(err)

    for rows.Next() {
        var uid int
        var username string
        var department string
        var created time.Time
        err = rows.Scan(&uid, &username, &department, &created)
        checkErr(err)
        fmt.Println(uid)
        fmt.Println(username)
        fmt.Println(department)
        fmt.Println(created)
    }

    //删除数据
    stmt, err = db.Prepare("delete from userinfo where uid=?")
    checkErr(err)
```

```
    res, err = stmt.Exec(id)
    checkErr(err)

    affect, err = res.RowsAffected()
    checkErr(err)

    fmt.Println(affect)

    db.Close()

}

func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}
```

我们可以看到上面的代码和MySQL例子里面的代码几乎是一模一样的，唯一改变的就是导入的驱动改变了，然后调用 `sql.Open` 是采用了SQLite的方式打开。

> sqlite管理工具：http://sqliteadmin.orbmu2k.de/
>
> 可以方便的新建数据库管理。

links

- 目录
- 上一节: 使用MySQL数据库
- 下一节: 使用PostgreSQL数据库

5.4 使用PostgreSQL数据库

PostgreSQL 是一个自由的对象-关系数据库服务器(数据库管理系统)，它在灵活的BSD-风格许可证下发行。它提供了相对其他开放源代码数据库系统(比如 MySQL 和 Firebird)，和对专有系统比如 Oracle、Sybase、IBM 的 DB2 和 Microsoft SQL Server的一种选择。

PostgreSQL和MySQL比较，它更加庞大一点，因为它是用来替代Oracle而设计的。所以在企业应用中采用PostgreSQL是一个明智的选择。

MySQL被Oracle收购之后正在逐步的封闭（自MySQL 5.5.31以后的所有版本将不再遵循GPL协议），鉴于此，将来我们也许会选择PostgreSQL而不是MySQL作为项目的后端数据库。

驱动

Go实现的支持PostgreSQL的驱动也很多，因为国外很多人在开发中使用了这个数据库。

- https://github.com/lib/pq 支持database/sql驱动，纯Go写的
- https://github.com/jbarham/gopgsqldriver 支持database/sql驱动，纯Go写的
- https://github.com/lxn/go-pgsql 支持database/sql驱动，纯Go写的

在下面的示例中我采用了第一个驱动，因为它目前使用的人最多，在github上也比较活跃。

实例代码

数据库建表语句：

```
CREATE TABLE userinfo
(
    uid serial NOT NULL,
    username character varying(100) NOT NULL,
    departname character varying(500) NOT NULL,
    Created date,
    CONSTRAINT userinfo_pkey PRIMARY KEY (uid)
)
WITH (OIDS=FALSE);

CREATE TABLE userdeatail
(
    uid integer,
    intro character varying(100),
    profile character varying(100)
)
WITH(OIDS=FALSE);
```

看下面这个Go如何操作数据库表数据:增删改查

package main

```
import (
    "database/sql"
    "fmt"
    _ "https://github.com/lib/pq"
)

func main() {
    db, err := sql.Open("postgres", "user=astaxie password=astaxie
    checkErr(err)

    //插入数据
    stmt, err := db.Prepare("INSERT INTO userinfo(username,departna
    checkErr(err)

    res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")
    checkErr(err)

    //pg不支持这个函数，因为他没有类似MySQL的自增ID
    id, err := res.LastInsertId()
    checkErr(err)

    fmt.Println(id)

    //更新数据
    stmt, err = db.Prepare("update userinfo set username=$1 where u
    checkErr(err)

    res, err = stmt.Exec("astaxieupdate", 1)
    checkErr(err)
```

```
    affect, err := res.RowsAffected()
    checkErr(err)

    fmt.Println(affect)

    //查询数据
    rows, err := db.Query("SELECT * FROM userinfo")
    checkErr(err)

    for rows.Next() {
        var uid int
        var username string
        var department string
        var created string
        err = rows.Scan(&uid, &username, &department, &created)
        checkErr(err)
        fmt.Println(uid)
        fmt.Println(username)
        fmt.Println(department)
        fmt.Println(created)
    }

    //删除数据
    stmt, err = db.Prepare("delete from userinfo where uid=$1")
    checkErr(err)

    res, err = stmt.Exec(1)
    checkErr(err)

    affect, err = res.RowsAffected()
    checkErr(err)

    fmt.Println(affect)

    db.Close()

}

func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}
```

从上面的代码我们可以看到，PostgreSQL是通过 `$1` , `$2` 这种方式来指定要传递的参数，而不是MySQL中的 `?` ，另外在sql.Open中的dsn信息的格式也与MySQL的驱动中的dsn格式不一样，所以在使用时请注意它们的差异。

还有pg不支持LastInsertId函数，因为PostgreSQL内部没有实现类似MySQL的自增ID返回，其他的代码几乎是一模一样。

links

- 目录
- 上一节: 使用SQLite数据库
- 下一节: 使用beedb库进行ORM开发

5.5 使用beedb库进行ORM开发

beedb是我开发的一个Go进行ORM操作的库，它采用了Go style方式对数据库进行操作，实现了struct到数据表记录的映射。beedb是一个十分轻量级的Go ORM框架，开发这个库的本意降低复杂的ORM学习曲线，尽可能在ORM的运行效率和功能之间寻求一个平衡，beedb是目前开源的Go ORM框架中实现比较完整的一个库，而且运行效率相当不错，功能也基本能满足需求。但是目前还不支持关系关联，这个是接下来版本升级的重点。

beedb是支持database/sql标准接口的ORM库，所以理论上来说，只要数据库驱动支持database/sql接口就可以无缝的接入beedb。目前我测试过的驱动包括下面几个：

Mysql:github.com/ziutek/mymysql/godrv[*]

Mysql:code.google.com/p/go-mysql-driver[*]

PostgreSQL:github.com/bmizerany/pq[*]

SQLite:github.com/mattn/go-sqlite3[*]

MS ADODB: github.com/mattn/go-adodb[*]

ODBC: bitbucket.org/miquella/mgodbc[*]

安装

beedb支持go get方式安装，是完全按照Go Style的方式来实现的。

```
go get github.com/astaxie/beedb
```

如何初始化

首先你需要import相应的数据库驱动包、database/sql标准接口包以及beedb包，如下所示：

```
import (
    "database/sql"
    "github.com/astaxie/beedb"
    _ "github.com/ziutek/mymysql/godrv"
)
```

导入必须的package之后,我们需要打开到数据库的链接，然后创建一个beedb对象（以MySQL为例)，如下所示

```
db, err := sql.Open("mymysql", "test/xiemengjun/123456")
if err != nil {
    panic(err)
}
orm := beedb.New(db)
```

beedb的New函数实际上应该有两个参数，第一个参数标准接口的db，第二个参数是使用的数据库引擎，如果你使用的数据库引擎是MySQL/Sqlite,那么第二个参数都可以省略。

如果你使用的数据库是SQLServer，那么初始化需要：

```
orm = beedb.New(db, "mssql")
```

如果你使用了PostgreSQL，那么初始化需要：

```
orm = beedb.New(db, "pg")
```

目前beedb支持打印调试，你可以通过如下的代码实现调试

```
beedb.OnDebug=true
```

接下来我们的例子采用前面的数据库表Userinfo，现在我们建立相应的struct

```
type Userinfo struct {
    Uid     int `PK` //如果表的主键不是id，那么需要加上pk注释，显式的说这个
    Username    string
    Departname  string
    Created     time.Time
}
```

> 注意一点，beedb针对驼峰命名会自动帮你转化成下划线字段，例如你定义了Struct名字为 `UserInfo`，那么转化成底层实现的时候是 `user_info`，字段命名也遵循该规则。

插入数据

下面的代码演示了如何插入一条记录，可以看到我们操作的是struct对象，而不是原生的sql语句，最后通过调用Save接口将数据保存到数据库。

```
var saveone Userinfo
saveone.Username = "Test Add User"
saveone.Departname = "Test Add Departname"
saveone.Created = time.Now()
orm.Save(&saveone)
```

我们看到插入之后 `saveone.Uid` 就是插入成功之后的自增ID。Save接口会自动帮你存进去。

beedb接口提供了另外一种插入的方式，map数据插入。

```
add := make(map[string]interface{})
add["username"] = "astaxie"
add["departname"] = "cloud develop"
add["created"] = "2012-12-02"
orm.SetTable("userinfo").Insert(add)
```

插入多条数据

```
addslice := make([]map[string]interface{}, 0)
add:=make(map[string]interface{})
add2:=make(map[string]interface{})
add["username"] = "astaxie"
add["departname"] = "cloud develop"
add["created"] = "2012-12-02"
add2["username"] = "astaxie2"
add2["departname"] = "cloud develop2"
add2["created"] = "2012-12-02"
addslice =append(addslice, add, add2)
orm.SetTable("userinfo").InsertBatch(addslice)
```

上面的操作方式有点类似链式查询，熟悉jquery的同学应该会觉得很亲切，每次调用的method都会返回原orm对象，以便可以继续调用该对象上的其他method。

上面我们调用的SetTable函数是显式的告诉ORM，我要执行的这个map对应的数据库表是 `userinfo` 。

更新数据

继续上面的例子来演示更新操作，现在saveone的主键已经有值了，此时调用save接口，beedb内部会自动调用update以进行数据的更新而非插入操作。

```
saveone.Username = "Update Username"
saveone.Departname = "Update Departname"
saveone.Created = time.Now()
orm.Save(&saveone)  //现在saveone有了主键值，就执行更新操作
```

更新数据也支持直接使用map操作

```
t := make(map[string]interface{})
t["username"] = "astaxie"
orm.SetTable("userinfo").SetPK("uid").Where(2).Update(t)
```

这里我们调用了几个beedb的函数

SetPK：显式的告诉ORM，数据库表 `userinfo` 的主键是 `uid` 。

Where:用来设置条件，支持多个参数，第一个参数如果为整数，相当于调用了Where("主键=?",值)。 Updata函数接收map类型的数据，执行更新数据。

查询数据

beedb的查询接口比较灵活，具体使用请看下面的例子

例子1，根据主键获取数据：

```
var user Userinfo
//Where接受两个参数，支持整形参数
orm.Where("uid=?", 27).Find(&user)
```

例子2：

```
var user2 Userinfo
orm.Where(3).Find(&user2) // 这是上面版本的缩写版，可以省略主键
```

例子3，不是主键类型的的条件：

```
var user3 Userinfo
//Where接受两个参数，支持字符型的参数
orm.Where("name     = ?", "john").Find(&user3)
```

例子4，更加复杂的条件：

```
var user4 Userinfo
//Where支持三个参数
orm.Where("name = ? and age < ?", "john", 88).Find(&user4)
```

可以通过如下接口获取多条数据，请看示例

例子1，根据条件id>3，获取20位置开始的10条数据的数据

```
var allusers []Userinfo
err := orm.Where("id > ?", "3").Limit(10,20).FindAll(&allusers)
```

例子2，省略limit第二个参数，默认从0开始，获取10条数据

```
var tenusers []Userinfo
err := orm.Where("id > ?", "3").Limit(10).FindAll(&tenusers)
```

例子3，获取全部数据

```
var everyone []Userinfo
err := orm.OrderBy("uid desc,username asc").FindAll(&everyone)
```

上面这些里面里面我们看到一个函数Limit，他是用来控制查询结构条数的。

Limit:支持两个参数，第一个参数表示查询的条数，第二个参数表示读取数据的起始位置，默认为0。

OrderBy:这个函数用来进行查询排序，参数是需要排序的条件。

上面这些例子都是将获取的的数据直接映射成struct对象，如果我们只是想获取一些数据到map，以下方式可以实现：

```
a, _ := orm.SetTable("userinfo").SetPK("uid").Where(2).Select("uid,
```

上面和这个例子里面又出现了一个新的接口函数Select，这个函数用来指定需要查询多少个字段。默认为全部字段 `*` 。

FindMap()函数返回的是 `[]map[string][]byte` 类型，所以你需要自己作类型转换。

删除数据

beedb提供了丰富的删除数据接口，请看下面的例子

例子1，删除单条数据

```
//saveone就是上面示例中的那个saveone
orm.Delete(&saveone)
```

例子2，删除多条数据

```
//alluser就是上面定义的获取多条数据的slice
orm.DeleteAll(&alluser)
```

例子3，根据sql删除数据

```
orm.SetTable("userinfo").Where("uid>?", 3).DeleteRow()
```

关联查询

目前beedb还不支持struct的关联关系，但是有些应用却需要用到连接查询，所以现在beedb提供了一个简陋的实现方案：

```
a, _ := orm.SetTable("userinfo").Join("LEFT", "userdeatail", "useri
```

上面代码中我们看到了一个新的接口Join函数，这个函数带有三个参数

- 第一个参数可以是：INNER, LEFT, OUTER, CROSS等
- 第二个参数表示连接的表
- 第三个参数表示连接的条件

Group By和Having

针对有些应用需要用到group by和having的功能，beedb也提供了一个简陋的实现

```
a, _ := orm.SetTable("userinfo").GroupBy("username").Having("userna
```

上面的代码中出现了两个新接口函数

GroupBy:用来指定进行groupby的字段

Having:用来指定having执行的时候的条件

进一步的发展

目前beedb已经获得了很多来自国内外用户的反馈，我目前也正在考虑重构，接下来会在几个方面进行改进

- 实现interface设计，类似databse/sql/driver的设计，设计beedb的接口，然后去实现相应数据库的CRUD操作
- 实现关联数据库设计，支持一对一，一对多，多对多的实现，示例代码如下：

```
type Profile struct{
    Nickname    string
    Mobile        string
}

type Userinfo struct {
    Uid     int `PK`
    Username    string
    Departname  string
    Created     time.Time
    Profile     `HasOne`
}
```

- 自动建库建表建索引
- 实现连接池的实现，采用goroutine

links

- 目录
- 上一节: 使用PostgreSQL数据库
- 下一节: NOSQL数据库操作

5.6 NOSQL数据库操作

NoSQL(Not Only SQL)，指的是非关系型的数据库。随着Web2.0的兴起，传统的关系数据库在应付Web2.0网站，特别是超大规模和高并发的SNS类型的Web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。

而Go语言作为21世纪的C语言，对NOSQL的支持也是很好，目前流行的NOSQL主要有redis、mongoDB、Cassandra和Membase等。这些数据库都有高性能、高并发读写等特点，目前已经广泛应用于各种应用中。我接下来主要讲解一下redis和mongoDB的操作。

redis

redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)和zset(有序集合)。

目前应用redis最广泛的应该是新浪微博平台，其次还有Facebook收购的图片社交网站instagram。以及其他一些有名的互联网企业

Go目前支持redis的驱动有如下

- https://github.com/alphazero/Go-Redis
- http://code.google.com/p/tideland-rdc/
- https://github.com/simonz05/godis
- https://github.com/hoisie/redis.go

目前我fork了最后一个驱动，更新了一些bug，目前应用在我自己的短域名服务项目中(每天200W左右的PV值)

https://github.com/astaxie/goredis

接下来的以我自己fork的这个redis驱动为例来演示如何进行数据的操作

```
package main

import (
    "github.com/astaxie/goredis"
    "fmt"
)

func main() {
    var client goredis.Client
    // 设置端口为redis默认端口
    client.Addr = "127.0.0.1:6379"

    //字符串操作
    client.Set("a", []byte("hello"))
    val, _ := client.Get("a")
    fmt.Println(string(val))
    client.Del("a")

    //list操作
    vals := []string{"a", "b", "c", "d", "e"}
    for _, v := range vals {
        client.Rpush("l", []byte(v))
    }
    dbvals,_ := client.Lrange("l", 0, 4)
    for i, v := range dbvals {
        println(i,":",string(v))
    }
    client.Del("l")
}
```

我们可以看到操作redis非常的方便，而且我实际项目中应用下来性能也很高。client的命令和redis的命令基本保持一致。所以和原生态操作redis非常类似。

mongoDB

MongoDB是一个高性能，开源，无模式的文档型数据库，是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，采用的是类似json的bjson格式来存储数据，因此可以存储比较复杂的数据类型。Mongo最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

下图展示了mysql和mongoDB之间的对应关系，我们可以看出来非常的方便，但是mongoDB的性能非常好。

图5.1 MongoDB和Mysql的操作对比图

目前Go支持mongoDB最好的驱动就是mgo，这个驱动目前最有可能成为官方的pkg。

下面我将演示如何通过Go来操作mongoDB：

```
package main

import (
    "fmt"
    "labix.org/v2/mgo"
    "labix.org/v2/mgo/bson"
)

type Person struct {
    Name string
    Phone string
}

func main() {
    session, err := mgo.Dial("server1.example.com,server2.example.c
    if err != nil {
        panic(err)
    }
    defer session.Close()

    session.SetMode(mgo.Monotonic, true)

    c := session.DB("test").C("people")
    err = c.Insert(&Person{"Ale", "+55 53 8116 9639"},
        &Person{"Cla", "+55 53 8402 8510"})
    if err != nil {
        panic(err)
    }

    result := Person{}
    err = c.Find(bson.M{"name": "Ale"}).One(&result)
    if err != nil {
        panic(err)
    }

    fmt.Println("Phone:", result.Phone)
}
```

我们可以看出来mgo的操作方式和beedb的操作方式几乎类似，都是基于struct的操作方式，这个就是Go Style。

links

- 目录

- 上一节: 使用beedb库进行ORM开发
- 下一节: 小结

5.7 小结

这一章我们讲解了Go如何设计database/sql接口，然后介绍了各种第三方关系型数据库驱动的使用。接着介绍了beedb，一种基于关系型数据库的ORM库，如何对数据库进行简单的操作。最后介绍了NOSQL的一些知识，目前Go对于NOSQL支持还是不错，因为Go作为21世纪的C语言，那么对于21世纪的数据库也是支持的相当好。

通过这一章的学习，我们学会了如何操作各种数据库，那么就解决了我们数据存储的问题，这是Web里面最重要的一部分，所以希望大家能够深入的去了解database/sql的设计思想。

> Go database/sql tutorial 里提供了惯用的范例及详细的说明。

links

- 目录
- 上一节: NOSQL数据库操作
- 下一章: session和数据存储

6 session和数据存储

Web开发中一个很重要的议题就是如何做好用户的整个浏览过程的控制，因为HTTP协议是无状态的，所以用户的每一次请求都是无状态的，我们不知道在整个Web操作过程中哪些连接与该用户有关，我们应该如何来解决这个问题呢？Web里面经典的解决方案是cookie和session，cookie机制是一种客户端机制，把用户数据保存在客户端，而session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构来保存信息，每一个网站访客都会被分配给一个唯一的标志符,即sessionID,它的存放形式无非两种:要么经过url传递,要么保存在客户端的cookies里.当然,你也可以将Session保存到数据库里,这样会更安全,但效率方面会有所下降。

6.1小节里面讲介绍session机制和cookie机制的关系和区别，6.2讲解Go语言如何来实现session，里面讲实现一个简易的session管理器，6.3小节讲解如何防止session被劫持的情况，如何有效的保护session。我们知道session其实可以存储在任何地方，6.3小节里面实现的session是存储在内存中的，但是如果我们的应用进一步扩展了，要实现应用的session共享，那么我们可以把session存储在数据库中(memcache或者redis)，6.4小节将详细的讲解如何实现这些功能。

目录

links

- 目录
- 上一章: 第五章总结
- 下一节: session和cookie

6.1 session和cookie

session和cookie是网站浏览中较为常见的两个概念，也是比较难以辨析的两个概念，但它们在浏览需要认证的服务页面以及页面统计中却相当关键。我们先来了解一下session和cookie怎么来的？考虑这样一个问题：

如何抓取一个访问受限的网页？如新浪微博好友的主页，个人微博页面等。

显然，通过浏览器，我们可以手动输入用户名和密码来访问页面，而所谓的“抓取”，其实就是使用程序来模拟完成同样的工作，因此我们需要了解“登陆”过程中到底发生了什么。

当用户来到微博登陆页面，输入用户名和密码之后点击“登录”后浏览器将认证信息POST给远端的服务器，服务器执行验证逻辑，如果验证通过，则浏览器会跳转到登录用户的微博首页，在登录成功后，服务器如何验证我们对其他受限制页面的访问呢？因为HTTP协议是无状态的，所以很显然服务器不可能知道我们已经在上一次的HTTP请求中通过了验证。当然，最简单的解决方案就是所有的请求里面都带上用户名和密码，这样虽然可行，但大大加重了服务器的负担（对于每个request都需要到数据库验证），也大大降低了用户体验(每个页面都需要重新输入用户名密码，每个页面都带有登录表单)。既然直接在请求中带上用户名与密码不可行，那么就只有在服务器或客户端保存一些类似的可以代表身份的信息了，所以就有了cookie与session。

cookie，简而言之就是在本地计算机保存一些用户操作的历史信息（当然包括登录信息），并在用户再次访问该站点时浏览器通过HTTP协议将本地cookie内容发送给服务器，从而完成验证，或继续上一步操作。

图6.1 cookie的原理图

session，简而言之就是在服务器上保存用户操作的历史信息。服务器使用session id来标识session，session id由服务器负责产生，保证随机性与唯一性，相当于一个随机密钥，避免在握手或传输中暴露用户真实密码。但该方式下，仍然需要将发送请求的客户端与session进行对应，所以可以借助cookie机制来获取客户端的标识（即session id），也可以通过GET方式将id提交给服务器。

图6.2 session的原理图

cookie

Cookie是由浏览器维持的，存储在客户端的一小段文本信息，伴随着用户请求和页面在Web服务器和浏览器之间传递。用户每次访问站点时，Web应用程序都可以读取cookie包含的信息。浏览器设置里面有cookie隐私数据选项，打开它，可以看到很多已访问网站的cookies，如下图所示：

图6.3 浏览器端保存的cookie信息

cookie是有时间限制的，根据生命期不同分成两种：会话cookie和持久cookie；

如果不设置过期时间，则表示这个cookie生命周期为从创建到浏览器关闭止，只要关闭浏览器窗口，cookie就消失了。这种生命期为浏览会话期的cookie被称为会话cookie。会话cookie一般不保存在硬盘上而是保存在内存里。

如果设置了过期时间(setMaxAge(606024))，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie依然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏览器进程间共享，比如两个IE窗口。而对于保存在内存的cookie，不同的浏览器有不同的处理方式。

Go设置cookie

Go语言中通过net/http包中的SetCookie来设置：

```
http.SetCookie(w ResponseWriter, cookie *Cookie)
```

w表示需要写入的response，cookie是一个struct，让我们来看一下cookie对象是怎么样的

```
type Cookie struct {
    Name       string
    Value      string
    Path       string
    Domain     string
    Expires    time.Time
    RawExpires string

// MaxAge=0 means no 'Max-Age' attribute specified.
// MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
// MaxAge>0 means Max-Age attribute present and given in seconds
    MaxAge   int
    Secure   bool
    HttpOnly bool
    Raw      string
    Unparsed []string // Raw text of unparsed attribute-value pairs
}
```

我们来看一个例子，如何设置cookie

```
expiration := time.Now()
expiration = expiration.AddDate(1, 0, 0)
cookie := http.Cookie{Name: "username", Value: "astaxie", Expires:
http.SetCookie(w, &cookie)
```

Go读取cookie

上面的例子演示了如何设置cookie数据，我们这里来演示一下如何读取cookie

```
cookie, _ := r.Cookie("username")
fmt.Fprint(w, cookie)
```

还有另外一种读取方式

```
for _, cookie := range r.Cookies() {
    fmt.Fprint(w, cookie.Name)
}
```

可以看到通过request获取cookie非常方便。

session

session，中文经常翻译为会话，其本来的含义是指有始有终的一系列动作/消息，比如打电话是从拿起电话拨号到挂断电话这中间的一系列过程可以称之为一个session。然而当session一词与网络协议相关联时，它又往往隐含了“面向连接”和/或“保持状态”这样两个含义。

session在Web开发环境下的语义又有了新的扩展，它的含义是指一类用来在客户端与服务器端之间保持状态的解决方案。有时候Session也用来指这种解决方案的存储结构。

session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构(也可能就是使用散列表)来保存信息。

但程序需要为某个客户端的请求创建一个session的时候，服务器首先检查这个客户端的请求里是否包含了一个session标识－称为session id，如果已经包含一个session id则说明以前已经为此客户创建过session，服务器就按照session id把这个session检索出来使用(如果检索不到，可能会新建一个，这种情况可能出现在服务端已经删除了该用户对应的session对象，但用户人为地在请求的URL后面附加上一个JSESSION的参数)。如果客户请求不包含session id，则为此客户创建一个session并且同时生成一个与此session相关联的session id，这个session id将在本次响应中返回给客户端保存。

session机制本身并不复杂，然而其实现和配置上的灵活性却使得具体情况复杂多变。这也要求我们不能把仅仅某一次的经验或者某一个浏览器，服务器的经验当作普遍适用的。

小结

如上文所述，session和cookie的目的相同，都是为了克服http协议无状态的缺陷，但完成的方法不同。session通过cookie，在客户端保存session id，而将用户的其他会话消息保存在服务端的session对象中，与此相对的，cookie需要将所有信息都保存在客户端。因此cookie存在着一定的安全隐患，例如本地cookie中保存的用户名密码被破译，或cookie被其他网站收集（例如：1. appA主动设置域B cookie，让域B cookie获取；2. XSS，在appA上通过javascript获取document.cookie，并传递给自己的appB）。

通过上面的一些简单介绍我们了解了cookie和session的一些基础知识，知道他们之间的联系和区别，做web开发之前，有必要将一些必要知识了解清楚，才不会在用到时捉襟见肘，或是在调bug时候如无头苍蝇乱转。接下来的几小节我们将详细介绍session相关的知识。

links

- 目录
- 上一节: session和数据存储
- 下一节: Go如何使用session

6.2 Go如何使用session

通过上一小节的介绍，我们知道session是在服务器端实现的一种用户和服务器之间认证的解决方案，目前Go标准包没有为session提供任何支持，这小节我们将会自己动手来实现go版本的session管理和创建。

session创建过程

session的基本原理是由服务器为每个会话维护一份信息数据，客户端和服务端依靠一个全局唯一的标识来访问这份数据，以达到交互的目的。当用户访问Web应用时，服务端程序会随需要创建session，这个过程可以概括为三个步骤：

- 生成全局唯一标识符（sessionid）；
- 开辟数据存储空间。一般会在内存中创建相应的数据结构，但这种情况下，系统一旦掉电，所有的会话数据就会丢失，如果是电子商务类网站，这将造成严重的后果。所以为了解决这类问题，你可以将会话数据写到文件里或存储在数据库中，当然这样会增加I/O开销，但是它可以实现某种程度的session持久化，也更有利于session的共享；
- 将session的全局唯一标示符发送给客户端。

以上三个步骤中，最关键的是如何发送这个session的唯一标识这一步上。考虑到HTTP协议的定义，数据无非可以放到请求行、头域或Body里，所以一般来说会有两种常用的方式：cookie和URL重写。

1. Cookie 服务端通过设置Set-cookie头就可以将session的标识符传送到客户端，而客户端此后的每一次请求都会带上这个标识符，另外一般包含session信息的cookie会将失效时间设置为0(会话cookie)，即浏览器进程有效时间。至于浏览器怎么处理这个0，每个浏览器都有自己的方案，但差别都不会太大(一般体现在新建浏览器窗口的时候)；
2. URL重写 所谓URL重写，就是在返回给用户的页面里的所有的URL后面追加session标识符，这样用户在收到响应之后，无论点击响应页面里的哪个链接或提交表单，都会自动带上session标识符，从而就实现了会话的保持。虽然这种做法比较麻烦，但是，如果客户端禁用了cookie的话，此种方案将会是首选。

Go实现session管理

通过上面session创建过程的讲解，读者应该对session有了一个大体的认识，但是具体到动态页面技术里面，又是怎么实现session的呢？下面我们将结合session的生命周期（lifecycle），来实现go语言版本的session管理。

session管理设计

我们知道session管理涉及到如下几个因素

- 全局session管理器
- 保证sessionid 的全局唯一性
- 为每个客户关联一个session
- session 的存储(可以存储到内存、文件、数据库等)
- session 过期处理

接下来我将讲解一下我关于session管理的整个设计思路以及相应的go代码示例：

Session管理器

定义一个全局的session管理器

```
type Manager struct {
    cookieName  string     //private cookiename
    lock        sync.Mutex // protects session
    provider    Provider
    maxlifetime int64
}

func NewManager(provideName, cookieName string, maxlifetime int64)
    provider, ok := provides[provideName]
    if !ok {
        return nil, fmt.Errorf("session: unknown provide %q (forgot
    }
    return &Manager{provider: provider, cookieName: cookieName, max
}
```

Go实现整个的流程应该也是这样的，在main包中创建一个全局的session管理器

```
var globalSessions *session.Manager
//然后在init函数中初始化
func init() {
    globalSessions, _ = NewManager("memory","gosessionid",3600)
}
```

我们知道session是保存在服务器端的数据，它可以以任何的方式存储，比如存储在内存、数据库或者文件中。因此我们抽象出一个Provider接口，用以表征session管理器底层存储结构。

```
type Provider interface {
    SessionInit(sid string) (Session, error)
    SessionRead(sid string) (Session, error)
    SessionDestroy(sid string) error
    SessionGC(maxLifeTime int64)
}
```

- SessionInit函数实现Session的初始化，操作成功则返回此新的Session变量
- SessionRead函数返回sid所代表的Session变量，如果不存在，那么将以sid为参数调用SessionInit函数创建并返回一个新的Session变量
- SessionDestroy函数用来销毁sid对应的Session变量
- SessionGC根据maxLifeTime来删除过期的数据

那么Session接口需要实现什么样的功能呢？有过Web开发经验的读者知道，对Session的处理基本就 设置值、读取值、删除值以及获取当前sessionID这四个操作，所以我们的Session接口也就实现这四个操作。

```
type Session interface {
    Set(key, value interface{}) error //set session value
    Get(key interface{}) interface{}  //get session value
    Delete(key interface{}) error     //delete session value
    SessionID() string                //back current sessionID
}
```

> 以上设计思路来源于database/sql/driver，先定义好接口，然后具体的存储session的结构实现相应的接口并注册后，相应功能这样就可以使用了，以下是用来随需注册存储session的结构的Register函数的实现。

```
var provides = make(map[string]Provider)

// Register makes a session provide available by the provided name
// If Register is called twice with the same name or if driver is n
// it panics.
func Register(name string, provider Provider) {
    if provider == nil {
        panic("session: Register provide is nil")
    }
    if _, dup := provides[name]; dup {
        panic("session: Register called twice for provide " + name)
    }
    provides[name] = provider
}
```

全局唯一的Session ID

Session ID是用来识别访问Web应用的每一个用户，因此必须保证它是全局唯一的（GUID），下面代码展示了如何满足这一需求：

```
func (manager *Manager) sessionId() string {
    b := make([]byte, 32)
    if _, err := io.ReadFull(rand.Reader, b); err != nil {
        return ""
    }
    return base64.URLEncoding.EncodeToString(b)
}
```

session创建

我们需要为每个来访用户分配或获取与他相关连的Session，以便后面根据Session信息来验证操作。SessionStart这个函数就是用来检测是否已经有某个Session与当前来访用户发生了关联，如果没有则创建之。

```
func (manager *Manager) SessionStart(w http.ResponseWriter, r *http
    manager.lock.Lock()
    defer manager.lock.Unlock()
    cookie, err := r.Cookie(manager.cookieName)
    if err != nil || cookie.Value == "" {
        sid := manager.sessionId()
        session, _ = manager.provider.SessionInit(sid)
        cookie := http.Cookie{Name: manager.cookieName, Value: url
        http.SetCookie(w, &cookie)
    } else {
        sid, _ := url.QueryUnescape(cookie.Value)
        session, _ = manager.provider.SessionRead(sid)
    }
    return
}
```

我们用前面login操作来演示session的运用：

```
func login(w http.ResponseWriter, r *http.Request) {
    sess := globalSessions.SessionStart(w, r)
    r.ParseForm()
    if r.Method == "GET" {
        t, _ := template.ParseFiles("login.gtpl")
        w.Header().Set("Content-Type", "text/html")
        t.Execute(w, sess.Get("username"))
    } else {
        sess.Set("username", r.Form["username"])
        http.Redirect(w, r, "/", 302)
    }
}
```

操作值：设置、读取和删除

SessionStart函数返回的是一个满足Session接口的变量，那么我们该如何用他来对session数据进行操作呢？

上面的例子中的代码 `session.Get("uid")` 已经展示了基本的读取数据的操作，现在我们再来看一下详细的操作:

```
func count(w http.ResponseWriter, r *http.Request) {
    sess := globalSessions.SessionStart(w, r)
    createtime := sess.Get("createtime")
    if createtime == nil {
        sess.Set("createtime", time.Now().Unix())
    } else if (createtime.(int64) + 360) < (time.Now().Unix()) {
        globalSessions.SessionDestroy(w, r)
        sess = globalSessions.SessionStart(w, r)
    }
    ct := sess.Get("countnum")
    if ct == nil {
        sess.Set("countnum", 1)
    } else {
        sess.Set("countnum", (ct.(int) + 1))
    }
    t, _ := template.ParseFiles("count.gtpl")
    w.Header().Set("Content-Type", "text/html")
    t.Execute(w, sess.Get("countnum"))
}
```

通过上面的例子可以看到，Session的操作和操作key/value数据库类似:Set、Get、Delete等操作

因为Session有过期的概念，所以我们定义了GC操作，当访问过期时间满足GC的触发条件后将会引起GC，但是当我们进行了任意一个session操作，都会对Session实体进行更新，都会触发对最后访问时间的修改，这样当GC的时候就不会误删除还在使用的Session实体。

session重置

我们知道，Web应用中有用户退出这个操作，那么当用户退出应用的时候，我们需要对该用户的session数据进行销毁操作，上面的代码已经演示了如何使用session重置操作，下面这个函数就是实现了这个功能：

```
//Destroy sessionid
func (manager *Manager) SessionDestroy(w http.ResponseWriter, r *ht
    cookie, err := r.Cookie(manager.cookieName)
    if err != nil || cookie.Value == "" {
        return
    } else {
        manager.lock.Lock()
        defer manager.lock.Unlock()
        manager.provider.SessionDestroy(cookie.Value)
        expiration := time.Now()
        cookie := http.Cookie{Name: manager.cookieName, Path: "/",
        http.SetCookie(w, &cookie)
    }
}
```

session销毁

我们来看一下Session管理器如何来管理销毁,只要我们在Main启动的时候启动：

```
func init() {
    go globalSessions.GC()
}

func (manager *Manager) GC() {
    manager.lock.Lock()
    defer manager.lock.Unlock()
    manager.provider.SessionGC(manager.maxlifetime)
    time.AfterFunc(time.Duration(manager.maxlifetime), func() { man
}
```

我们可以看到GC充分利用了time包中的定时器功能，当超时 `maxLifeTime` 之后调用GC函数，这样就可以保证 `maxLifeTime` 时间内的session都是可用的，类似的方案也可以用于统计在线用户数之类的。

总结

至此 我们实现了一个用来在Web应用中全局管理Session的SessionManager，定义了用来提供Session存储实现Provider的接口,下一小节，我们将会通过接口定义来实现一些Provider,供大家参考学习。

links

- 目录

- 上一节: session和cookie
- 下一节: session存储

6.3 session存储

上一节我们介绍了Session管理器的实现原理，定义了存储session的接口，这小节我们将示例一个基于内存的session存储接口的实现，其他的存储方式，读者可以自行参考示例来实现，内存的实现请看下面的例子代码

```
package memory

import (
    "container/list"
    "github.com/astaxie/session"
    "sync"
    "time"
)

var pder = &Provider{list: list.New()}

type SessionStore struct {
    sid          string                      //session id唯一标示
    timeAccessed time.Time                   //最后访问时间
    value        map[interface{}]interface{} //session里面存储的值
}

func (st *SessionStore) Set(key, value interface{}) error {
    st.value[key] = value
    pder.SessionUpdate(st.sid)
    return nil
}

func (st *SessionStore) Get(key interface{}) interface{} {
    pder.SessionUpdate(st.sid)
    if v, ok := st.value[key]; ok {
        return v
    } else {
        return nil
    }
    return nil
}

func (st *SessionStore) Delete(key interface{}) error {
    delete(st.value, key)
    pder.SessionUpdate(st.sid)
    return nil
}

func (st *SessionStore) SessionID() string {
    return st.sid
}
```

```
type Provider struct {
    lock     sync.Mutex               //用来锁
    sessions map[string]*list.Element //用来存储在内存
    list     *list.List               //用来做gc
}

func (pder *Provider) SessionInit(sid string) (session.Session, err
    pder.lock.Lock()
    defer pder.lock.Unlock()
    v := make(map[interface{}]interface{}, 0)
    newsess := &SessionStore{sid: sid, timeAccessed: time.Now(), va
    element := pder.list.PushBack(newsess)
    pder.sessions[sid] = element
    return newsess, nil
}

func (pder *Provider) SessionRead(sid string) (session.Session, err
    if element, ok := pder.sessions[sid]; ok {
        return element.Value.(*SessionStore), nil
    } else {
        sess, err := pder.SessionInit(sid)
        return sess, err
    }
    return nil, nil
}

func (pder *Provider) SessionDestroy(sid string) error {
    if element, ok := pder.sessions[sid]; ok {
        delete(pder.sessions, sid)
        pder.list.Remove(element)
        return nil
    }
    return nil
}

func (pder *Provider) SessionGC(maxlifetime int64) {
    pder.lock.Lock()
    defer pder.lock.Unlock()

    for {
        element := pder.list.Back()
        if element == nil {
            break
        }
        if (element.Value.(*SessionStore).timeAccessed.Unix() + max
            pder.list.Remove(element)
            delete(pder.sessions, element.Value.(*SessionStore).sid
        } else {
            break
        }
    }
}
```

```
func (pder *Provider) SessionUpdate(sid string) error {
    pder.lock.Lock()
    defer pder.lock.Unlock()
    if element, ok := pder.sessions[sid]; ok {
        element.Value.(*SessionStore).timeAccessed = time.Now()
        pder.list.MoveToFront(element)
        return nil
    }
    return nil
}

func init() {
    pder.sessions = make(map[string]*list.Element, 0)
    session.Register("memory", pder)
}
```

上面这个代码实现了一个内存存储的session机制。通过init函数注册到session管理器中。这样就可以方便的调用了。我们如何来调用该引擎呢？请看下面的代码

```
import (
    "github.com/astaxie/session"
    _ "github.com/astaxie/session/providers/memory"
)
```

当import的时候已经执行了memory函数里面的init函数，这样就已经注册到session管理器中，我们就可以使用了，通过如下方式就可以初始化一个session管理器：

```
var globalSessions *session.Manager

//然后在init函数中初始化
func init() {
    globalSessions, _ = session.NewManager("memory", "gosessionid",
    go globalSessions.GC()
}
```

links

- 目录
- 上一节: Go如何使用session
- 下一节: 预防session劫持

6.4 预防session劫持

session劫持是一种广泛存在的比较严重的安全威胁，在session技术中，客户端和服务端通过session的标识符来维护会话， 但这个标识符很容易就能被嗅探到，从而被其他人利用.它是中间人攻击的一种类型。

本节将通过一个实例来演示会话劫持，希望通过这个实例，能让读者更好地理解session的本质。

session劫持过程

我们写了如下的代码来展示一个count计数器：

```
func count(w http.ResponseWriter, r *http.Request) {
    sess := globalSessions.SessionStart(w, r)
    ct := sess.Get("countnum")
    if ct == nil {
        sess.Set("countnum", 1)
    } else {
        sess.Set("countnum", (ct.(int) + 1))
    }
    t, _ := template.ParseFiles("count.gtpl")
    w.Header().Set("Content-Type", "text/html")
    t.Execute(w, sess.Get("countnum"))
}
```

count.gtpl的代码如下所示：

```
Hi. Now count:{{.}}
```

然后我们在浏览器里面刷新可以看到如下内容：

图6.4 浏览器端显示count数

随着刷新，数字将不断增长，当数字显示为6的时候，打开浏览器(以chrome为例）的cookie管理器，可以看到类似如下的信息：

图6.5 获取浏览器端保存的cookie

下面这个步骤最为关键: 打开另一个浏览器(这里我打开了firefox浏览器),复制chrome地址栏里的地址到新打开的浏览器的地址栏中。然后打开firefox的cookie模拟插件，新建一个cookie，把按上图中cookie内容原样在firefox中重建一份:

图6.6 模拟cookie

回车后，你将看到如下内容：

图6.7 劫持session成功

可以看到虽然换了浏览器，但是我们却获得了sessionID，然后模拟了cookie存储的过程。这个例子是在同一台计算机上做的，不过即使换用两台来做，其结果仍然一样。此时如果交替点击两个浏览器里的链接你会发现它们其实操纵的是同一个计数器。不必惊讶，此处firefox盗用了chrome和goserver之间的维持会话的钥匙，即gosessionid，这是一种类型的“会话劫持”。在goserver看来，它从http请求中得到了一个gosessionid，由于HTTP协议的无状态性，它无法得知这个gosessionid是从chrome那里“劫持”来的，它依然会去查找对应的session，并执行相关计算。与此同时 chrome也无法得知自己保持的会话已经被“劫持”。

session劫持防范

cookieonly和token

通过上面session劫持的简单演示可以了解到session一旦被其他人劫持，就非常危险，劫持者可以假装成被劫持者进行很多非法操作。那么如何有效的防止session劫持呢？

其中一个解决方案就是sessionID的值只允许cookie设置，而不是通过URL重置方式设置，同时设置cookie的httponly为true,这个属性是设置是否可通过客户端脚本访问这个设置的cookie，第一这个可以防止这个cookie被XSS读取从而引起session劫持，第二cookie设置不会像URL重置方式那么容易获取sessionID。

第二步就是在每个请求里面加上token，实现类似前面章节里面讲的防止form重复递交类似的功能，我们在每个请求里面加上一个隐藏的token，然后每次验证这个token，从而保证用户的请求都是唯一性。

```
h := md5.New()
salt:="astaxie%^7&8888"
io.WriteString(h,salt+time.Now().String())
token:=fmt.Sprintf("%x",h.Sum(nil))
if r.Form["token"]!=token{
    //提示登录
}
sess.Set("token",token)
```

间隔生成新的SID

还有一个解决方案就是，我们给session额外设置一个创建时间的值，一旦过了一定的时间，我们销毁这个sessionID，重新生成新的session，这样可以一定程度上防止session劫持的问题。

```
createtime := sess.Get("createtime")
if createtime == nil {
    sess.Set("createtime", time.Now().Unix())
} else if (createtime.(int64) + 60) < (time.Now().Unix()) {
    globalSessions.SessionDestroy(w, r)
    sess = globalSessions.SessionStart(w, r)
}
```

session启动后，我们设置了一个值，用于记录生成sessionID的时间。通过判断每次请求是否过期(这里设置了60秒)定期生成新的ID，这样使得攻击者获取有效sessionID的机会大大降低。

上面两个手段的组合可以在实践中消除session劫持的风险，一方面， 由于sessionID频繁改变，使攻击者难有机会获取有效的sessionID；另一方面，因为sessionID只能在cookie中传递，然后设置了httponly，所以基于URL攻击的可能性为零，同时被XSS获取sessionID也不可能。最后，由于我们还设置了MaxAge=0，这样就相当于session cookie不会留在浏览器的历史记录里面。

links

- 目录
- 上一节: session存储
- 下一节: 小结

6.5 小结

这章我们学习了什么是session，什么是cookie，以及他们两者之间的关系。但是目前Go官方标准包里面不支持session，所以我们设计了一个session管理器，实现了session从创建到销毁的整个过程。然后定义了Provider的接口，使得可以支持各种后端的session存储，然后我们在第三小节里面介绍了如何使用内存存储来实现session的管理。第四小节我们讲解了session劫持的过程，以及我们如何有效的来防止session劫持。通过这一章的讲解，希望能够让读者了解整个sesison的执行原理以及如何实现，而且是如何更加安全的使用session。

links

- 目录
- 上一节: session存储
- 下一章: 文本处理

7 文本处理

Web开发中对于文本处理是非常重要的一部分，我们往往需要对输出或者输入的内容进行处理，这里的文本包括字符串、数字、Json、XMl等等。Go语言作为一门高性能的语言，对这些文本的处理都有官方的标准库来支持。而且在你使用中你会发现Go标准库的一些设计相当的巧妙，而且对于使用者来说也很方便就能处理这些文本。本章我们将通过四个小节的介绍，让用户对Go语言处理文本有一个很好的认识。

XML是目前很多标准接口的交互语言，很多时候和一些Java编写的webserver进行交互都是基于XML标准进行交互，7.1小节将介绍如何处理XML文本，我们使用XML之后发现它太复杂了，现在很多互联网企业对外的API大多数采用了JSON格式，这种格式描述简单，但是又能很好的表达意思，7.2小节我们将讲述如何来处理这样的JSON格式数据。正则是一个让人又爱又恨的工具，它处理文本的能力非常强大，我们在前面表单验证里面已经有所领略它的强大，7.3小节将详细的更深入的讲解如何利用好Go的正则。Web开发中一个很重要的部分就是MVC分离，在Go语言的Web开发中V有一个专门的包来支持`template`,7.4小节将详细的讲解如何使用模版来进行输出内容。7.5小节将详细介绍如何进行文件和文件夹的操作。7.6小结介绍了字符串的相关操作。

目录

links

- 目录
- 上一章: 第六章总结
- 下一节: XML处理

7.1 XML处理

XML作为一种数据交换和信息传递的格式已经十分普及。而随着Web服务日益广泛的应用，现在XML在日常的开发工作中也扮演了愈发重要的角色。这一小节， 我们将就Go语言标准包中的XML相关处理的包进行介绍。

这个小节不会涉及XML规范相关的内容（如需了解相关知识请参考其他文献），而是介绍如何用Go语言来编解码XML文件相关的知识。

假如你是一名运维人员，你为你所管理的所有服务器生成了如下内容的xml的配置文件：

```
<?xml version="1.0" encoding="utf-8"?>
<servers version="1">
    <server>
        <serverName>Shanghai_VPN</serverName>
        <serverIP>127.0.0.1</serverIP>
    </server>
    <server>
        <serverName>Beijing_VPN</serverName>
        <serverIP>127.0.0.2</serverIP>
    </server>
</servers>
```

上面的XML文档描述了两个服务器的信息，包含了服务器名和服务器的IP信息，接下来的Go例子以此XML描述的信息进行操作。

解析XML

如何解析如上这个XML文件呢？ 我们可以通过xml包的 `Unmarshal` 函数来达到我们的目的

```
func Unmarshal(data []byte, v interface{}) error
```

data接收的是XML数据流，v是需要输出的结构，定义为interface，也就是可以把XML转换为任意的格式。我们这里主要介绍struct的转换，因为struct和XML都有类似树结构的特征。

示例代码如下：

```
package main

import (
    "encoding/xml"
    "fmt"
    "io/ioutil"
    "os"
)

type Recurlyservers struct {
    XMLName     xml.Name `xml:"servers"`
    Version     string   `xml:"version,attr"`
    Svs         []server `xml:"server"`
    Description string   `xml:",innerxml"`
}

type server struct {
    XMLName    xml.Name `xml:"server"`
    ServerName string   `xml:"serverName"`
    ServerIP   string   `xml:"serverIP"`
}

func main() {
    file, err := os.Open("servers.xml") // For read access.
    if err != nil {
        fmt.Printf("error: %v", err)
        return
    }
    defer file.Close()
    data, err := ioutil.ReadAll(file)
    if err != nil {
        fmt.Printf("error: %v", err)
        return
    }
    v := Recurlyservers{}
    err = xml.Unmarshal(data, &v)
    if err != nil {
        fmt.Printf("error: %v", err)
        return
    }

    fmt.Println(v)
}
```

XML本质上是一种树形的数据格式，而我们可以定义与之匹配的go 语言的 struct类型，然后通过xml.Unmarshal来将xml中的数据解析成对应的struct对象。如上例子输出如下数据

```
{{ servers} 1 [{{ server} Shanghai_VPN 127.0.0.1} {{ server} Beijin
<server>
    <serverName>Shanghai_VPN</serverName>
    <serverIP>127.0.0.1</serverIP>
</server>
<server>
    <serverName>Beijing_VPN</serverName>
    <serverIP>127.0.0.2</serverIP>
</server>
}
```

上面的例子中，将xml文件解析成对应的struct对象是通过 `xml.Unmarshal` 来完成的，这个过程是如何实现的？可以看到我们的struct定义后面多了一些类似于 `xml:"serverName"` 这样的内容,这个是struct的一个特性，它们被称为 struct tag，它们是用来辅助反射的。我们来看一下 `Unmarshal` 的定义：

```
func Unmarshal(data []byte, v interface{}) error
```

我们看到函数定义了两个参数，第一个是XML数据流，第二个是存储的对应类型，目前支持struct、slice和string，XML包内部采用了反射来进行数据的映射，所以v里面的字段必须是导出的。 `Unmarshal` 解析的时候XML元素和字段怎么对应起来的呢？这是有一个优先级读取流程的，首先会读取struct tag，如果没有，那么就会对应字段名。必须注意一点的是解析的时候tag、字段名、XML元素都是大小写敏感的，所以必须一一对应字段。

Go语言的反射机制，可以利用这些tag信息来将来自XML文件中的数据反射成对应的struct对象，关于反射如何利用struct tag的更多内容请参阅reflect中的相关内容。

解析XML到struct的时候遵循如下的规则：

- 如果struct的一个字段是string或者[]byte类型且它的tag含有 `",innerxml"` ，Unmarshal将会将此字段所对应的元素内所有内嵌的原始xml累加到此字段上，如上面例子Description定义。最后的输出是

  ```
  <server>
      <serverName>Shanghai_VPN</serverName>
      <serverIP>127.0.0.1</serverIP>
  </server>
  <server>
      <serverName>Beijing_VPN</serverName>
      <serverIP>127.0.0.2</serverIP>
  </server>
  ```

- 如果struct中有一个叫做XMLName，且类型为xml.Name字段，那么在解析的时候就会保存这个element的名字到该字段,如上面例子中的servers。

- 如果某个struct字段的tag定义中含有XML结构中element的名称，那么解析的时候就会把相应的element值赋值给该字段，如上servername和serverip定义。
- 如果某个struct字段的tag定义了中含有 `",attr"` ，那么解析的时候就会将该结构所对应的element的与字段同名的属性的值赋值给该字段，如上version定义。
- 如果某个struct字段的tag定义 型如 `"a>b>c"` ,则解析的时候，会将xml结构a下面的b下面的c元素的值赋值给该字段。
- 如果某个struct字段的tag定义了 `"-"` ,那么不会为该字段解析匹配任何xml数据。
- 如果struct字段后面的tag定义了 `",any"` ，如果他的子元素在不满足其他的规则的时候就会匹配到这个字段。
- 如果某个XML元素包含一条或者多条注释，那么这些注释将被累加到第一个tag含有",comments"的字段上，这个字段的类型可能是[]byte或string,如果没有这样的字段存在，那么注释将会被抛弃。

上面详细讲述了如何定义struct的tag。 只要设置对了tag，那么XML解析就如上面示例般简单，tag和XML的element是一一对应的关系，如上所示，我们还可以通过slice来表示多个同级元素。

> 注意： 为了正确解析，go语言的xml包要求struct定义中的所有字段必须是可导出的（即首字母大写）

输出XML

假若我们不是要解析如上所示的XML文件，而是生成它，那么在go语言中又该如何实现呢？ xml包中提供了 `Marshal` 和 `MarshalIndent` 两个函数，来满足我们的需求。这两个函数主要的区别是第二个函数会增加前缀和缩进，函数的定义如下所示：

```
func Marshal(v interface{}) ([]byte, error)
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, e
```

两个函数第一个参数是用来生成XML的结构定义类型数据，都是返回生成的XML数据流。

下面我们来看一下如何输出如上的XML：

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

type Servers struct {
    XMLName xml.Name `xml:"servers"`
    Version string   `xml:"version,attr"`
    Svs     []server `xml:"server"`
}

type server struct {
    ServerName string `xml:"serverName"`
    ServerIP   string `xml:"serverIP"`
}

func main() {
    v := &Servers{Version: "1"}
    v.Svs = append(v.Svs, server{"Shanghai_VPN", "127.0.0.1"})
    v.Svs = append(v.Svs, server{"Beijing_VPN", "127.0.0.2"})
    output, err := xml.MarshalIndent(v, "  ", "    ")
    if err != nil {
        fmt.Printf("error: %v\n", err)
    }
    os.Stdout.Write([]byte(xml.Header))

    os.Stdout.Write(output)
}
```

上面的代码输出如下信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<servers version="1">
<server>
    <serverName>Shanghai_VPN</serverName>
    <serverIP>127.0.0.1</serverIP>
</server>
<server>
    <serverName>Beijing_VPN</serverName>
    <serverIP>127.0.0.2</serverIP>
</server>
</servers>
```

和我们之前定义的文件的格式一模一样，之所以会有 `os.Stdout.Write([]byte(xml.Header))` 这句代码的出现，是因为 `xml.MarshalIndent` 或者 `xml.Marshal` 输出的信息都是不带XML头的，为了

生成正确的xml文件，我们使用了xml包预定义的Header变量。

我们看到 `Marshal` 函数接收的参数v是interface{}类型的，即它可以接受任意类型的参数，那么xml包，根据什么规则来生成相应的XML文件呢？

- 如果v是 array或者slice，那么输出每一个元素，类似value
- 如果v是指针，那么会Marshal指针指向的内容，如果指针为空，什么都不输出
- 如果v是interface，那么就处理interface所包含的数据
- 如果v是其他数据类型，就会输出这个数据类型所拥有的字段信息

生成的XML文件中的element的名字又是根据什么决定的呢？元素名按照如下优先级从struct中获取：

- 如果v是struct，XMLName的tag中定义的名称
- 类型为xml.Name的名叫XMLName的字段的值
- 通过struct中字段的tag来获取
- 通过struct的字段名用来获取
- marshall的类型名称

我们应如何设置struct 中字段的tag信息以控制最终xml文件的生成呢？

- XMLName不会被输出
- tag中含有 `"-"` 的字段不会输出
- tag中含有 `"name,attr"` ，会以name作为属性名，字段值作为值输出为这个XML元素的属性，如上version字段所描述
- tag中含有 `",attr"` ，会以这个struct的字段名作为属性名输出为XML元素的属性，类似上一条，只是这个name默认是字段名了。
- tag中含有 `",chardata"` ，输出为xml的 character data而非element。
- tag中含有 `",innerxml"` ，将会被原样输出，而不会进行常规的编码过程
- tag中含有 `",comment"` ，将被当作xml注释来输出，而不会进行常规的编码过程，字段值中不能含有"--"字符串
- tag中含有 `"omitempty"` ,如果该字段的值为空值那么该字段就不会被输出到XML，空值包括：false、0、nil指针或nil接口，任何长度为0的array, slice, map或者string
- tag中含有 `"a>b>c"` ，那么就会循环输出三个元素a包含b，b包含c，例如如下代码就会输出

```
FirstName string   `xml:"name>first"`
LastName  string   `xml:"name>last"`

<name>
<first>Asta</first>
<last>Xie</last>
</name>
```

上面我们介绍了如何使用Go语言的xml包来编/解码XML文件，重要的一点是对XML的所有操作都是通过struct tag来实现的，所以学会对struct tag的运用变得非常重要，在文章中我们简要的列举了如何定义tag。更多内容或tag定义请参看相应的官方资料。

links

- 目录
- 上一节: 文本处理
- 下一节: Json处理

7.2 JSON处理

JSON（Javascript Object Notation）是一种轻量级的数据交换语言，以文字为基础，具有自我描述性且易于让人阅读。尽管JSON是Javascript的一个子集，但JSON是独立于语言的文本格式，并且采用了类似于C语言家族的一些习惯。JSON与XML最大的不同在于XML是一个完整的标记语言，而JSON不是。JSON由于比XML更小、更快，更易解析,以及浏览器的内建快速解析支持,使得其更适用于网络数据传输领域。目前我们看到很多的开放平台，基本上都是采用了JSON作为他们的数据交互的接口。既然JSON在Web开发中如此重要，那么Go语言对JSON支持的怎么样呢？Go语言的标准库已经非常好的支持了JSON，可以很容易的对JSON数据进行编、解码的工作。

前一小节的运维的例子用json来表示，结果描述如下：

```
{"servers":[{"serverName":"Shanghai_VPN","serverIP":"127.0.0.1"},{"
```

本小节余下的内容将以此JSON数据为基础，来介绍go语言的json包对JSON数据的编、解码。

解析JSON

解析到结构体

假如有了上面的JSON串，那么我们如何来解析这个JSON串呢？Go的JSON包中有如下函数

```
func Unmarshal(data []byte, v interface{}) error
```

通过这个函数我们就可以实现解析的目的，详细的解析例子请看如下代码：

```
package main

import (
    "encoding/json"
    "fmt"
)

type Server struct {
    ServerName string
    ServerIP   string
}

type Serverslice struct {
    Servers []Server
}

func main() {
    var s Serverslice
    str := `{"servers":[{"serverName":"Shanghai_VPN","serverIP":"12
    json.Unmarshal([]byte(str), &s)
    fmt.Println(s)
}
```

在上面的示例代码中，我们首先定义了与json数据对应的结构体，数组对应slice，字段名对应JSON里面的KEY，在解析的时候，如何将json数据与struct字段相匹配呢？例如JSON的key是 `Foo` ，那么怎么找对应的字段呢？

- 首先查找tag含有 `Foo` 的可导出的struct字段(首字母大写)
- 其次查找字段名是 `Foo` 的导出字段
- 最后查找类似 `FOO` 或者 `FoO` 这样的除了首字母之外其他大小写不敏感的导出字段

聪明的你一定注意到了这一点：能够被赋值的字段必须是可导出字段(即首字母大写）。同时JSON解析的时候只会解析能找得到的字段，找不到的字段会被忽略，这样的一个好处是：当你接收到一个很大的JSON数据结构而你却只想获取其中的部分数据的时候，你只需将你想要的数据对应的字段名大写，即可轻松解决这个问题。

解析到interface

上面那种解析方式是在我们知晓被解析的JSON数据的结构的前提下采取的方案，如果我们不知道被解析的数据的格式，又应该如何来解析呢？

我们知道interface{}可以用来存储任意数据类型的对象，这种数据结构正好用于存储解析的未知结构的json数据的结果。JSON包中采用map[string]interface{}和[]interface{}结构来存储任意的JSON对象和数组。Go类型和JSON类型的对应关系如下：

- bool 代表 JSON booleans,
- float64 代表 JSON numbers,
- string 代表 JSON strings,
- nil 代表 JSON null.

现在我们假设有如下的JSON数据

```
b := []byte(`{"Name":"Wednesday","Age":6,"Parents":["Gomez","Mortic
```

如果在我们不知道他的结构的情况下，我们把他解析到interface{}里面

```
var f interface{}
err := json.Unmarshal(b, &f)
```

这个时候f里面存储了一个map类型，他们的key是string，值存储在空的interface{}里

```
f = map[string]interface{}{
    "Name": "Wednesday",
    "Age":  6,
    "Parents": []interface{}{
        "Gomez",
        "Morticia",
    },
}
```

那么如何来访问这些数据呢？通过断言的方式：

```
m := f.(map[string]interface{})
```

通过断言之后，你就可以通过如下方式来访问里面的数据了

```
for k, v := range m {
    switch vv := v.(type) {
    case string:
        fmt.Println(k, "is string", vv)
    case int:
        fmt.Println(k, "is int", vv)
    case float64:
        fmt.Println(k,"is float64",vv)
    case []interface{}:
        fmt.Println(k, "is an array:")
        for i, u := range vv {
            fmt.Println(i, u)
        }
    default:
        fmt.Println(k, "is of a type I don't know how to handle")
    }
}
```

通过上面的示例可以看到，通过interface{}与type assert的配合，我们就可以解析未知结构的JSON数了。

上面这个是官方提供的解决方案，其实很多时候我们通过类型断言，操作起来不是很方便，目前bitly公司开源了一个叫做 `simplejson` 的包,在处理未知结构体的JSON时相当方便，详细例子如下所示：

```
js, err := NewJson([]byte(`{
    "test": {
        "array": [1, "2", 3],
        "int": 10,
        "float": 5.150,
        "bignum": 9223372036854775807,
        "string": "simplejson",
        "bool": true
    }
}`))

arr, _ := js.Get("test").Get("array").Array()
i, _ := js.Get("test").Get("int").Int()
ms := js.Get("test").Get("string").MustString()
```

可以看到，使用这个库操作JSON比起官方包来说，简单的多,详细的请参考如下地址：https://github.com/bitly/go-simplejson

生成JSON

我们开发很多应用的时候，最后都是要输出JSON数据串，那么如何来处理呢？JSON包里面通过 `Marshal` 函数来处理，函数定义如下：

```
func Marshal(v interface{}) ([]byte, error)
```

假设我们还是需要生成上面的服务器列表信息，那么如何来处理呢？请看下面的例子：

```
package main

import (
    "encoding/json"
    "fmt"
)

type Server struct {
    ServerName string
    ServerIP   string
}

type Serverslice struct {
    Servers []Server
}

func main() {
    var s Serverslice
    s.Servers = append(s.Servers, Server{ServerName: "Shanghai_VPN'
    s.Servers = append(s.Servers, Server{ServerName: "Beijing_VPN",
    b, err := json.Marshal(s)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(string(b))
}
```

输出如下内容：

```
{"Servers":[{"ServerName":"Shanghai_VPN","ServerIP":"127.0.0.1"},{'
```

我们看到上面的输出字段名的首字母都是大写的，如果你想用小写的首字母怎么办呢？把结构体的字段名改成首字母小写的？JSON输出的时候必须注意，只有导出的字段才会被输出，如果修改字段名，那么就会发现什么都不会输出，所以必须通过struct tag定义来实现：

```
type Server struct {
    ServerName string `json:"serverName"`
    ServerIP   string `json:"serverIP"`
}

type Serverslice struct {
    Servers []Server `json:"servers"`
}
```

通过修改上面的结构体定义，输出的JSON串就和我们最开始定义的JSON串保持一致了。

针对JSON的输出，我们在定义struct tag的时候需要注意的几点是:

- 字段的tag是 `"-"` ，那么这个字段不会输出到JSON
- tag中带有自定义名称，那么这个自定义名称会出现在JSON的字段名中，例如上面例子中serverName
- tag中如果带有 `"omitempty"` 选项，那么如果该字段值为空，就不会输出到JSON串中
- 如果字段类型是bool, string, int, int64等，而tag中带有 `",string"` 选项，那么这个字段在输出到JSON的时候会把该字段对应的值转换成JSON字符串

举例来说：

```
type Server struct {
    // ID 不会导出到JSON中
    ID int `json:"-"`

    // ServerName 的值会进行二次JSON编码
    ServerName  string `json:"serverName"`
    ServerName2 string `json:"serverName2,string"`

    // 如果 ServerIP 为空，则不输出到JSON串中
    ServerIP   string `json:"serverIP,omitempty"`
}

s := Server {
    ID:         3,
    ServerName:  `Go "1.0" `,
    ServerName2: `Go "1.0" `,
    ServerIP:   ``,
}
b, _ := json.Marshal(s)
os.Stdout.Write(b)
```

会输出以下内容：

```
{"serverName":"Go \"1.0\" ","serverName2":"\"Go \\\"1.0\\\" \""}
```

Marshal函数只有在转换成功的时候才会返回数据，在转换的过程中我们需要注意几点：

- JSON对象只支持string作为key，所以要编码一个map，那么必须是map[string]T这种类型(T是Go语言中任意的类型)
- Channel, complex和function是不能被编码成JSON的
- 嵌套的数据是不能编码的，不然会让JSON编码进入死循环
- 指针在编码的时候会输出指针指向的内容，而空指针会输出null

本小节，我们介绍了如何使用Go语言的json标准包来编解码JSON数据，同时也简要介绍了如何使用第三方包 `go-simplejson` 来在一些情况下简化操作，学会并熟练运用它们将对我们接下来的Web开发相当重要。

links

- 目录
- 上一节: XML处理
- 下一节: 正则处理

7.3 正则处理

正则表达式是一种进行模式匹配和文本操纵的复杂而又强大的工具。虽然正则表达式比纯粹的文本匹配效率低，但是它却更灵活。按照它的语法规则，随需构造出的匹配模式就能够从原始文本中筛选出几乎任何想你要得到的字符组合。如果你在Web开发中需要从一些文本数据源中获取数据,那么你只需要按照它的语法规则，随需构造出正确的模式字符串就能够从原数据源提取出有意义的文本信息。

Go语言通过 `regexp` 标准包为正则表达式提供了官方支持，如果你已经使用过其他编程语言提供的正则相关功能，那么你应该对Go语言版本的不会太陌生，但是它们之间也有一些小的差异，因为Go实现的是RE2标准，除了\C，详细的语法描述参考： `http://code.google.com/p/re2/wiki/Syntax`

其实字符串处理我们可以使用 `strings` 包来进行搜索(Contains、Index)、替换(Replace)和解析(Split、Join)等操作，但是这些都是简单的字符串操作，他们的搜索都是大小写敏感，而且固定的字符串，如果我们需要匹配可变的那种就没办法实现了，当然如果 `strings` 包能解决你的问题，那么就尽量使用它来解决。因为他们足够简单、而且性能和可读性都会比正则好。

如果你还记得，在前面表单验证的小节里，我们已经接触过正则处理，在那里我们利用了它来验证输入的信息是否满足某些预设的条件。在使用中需要注意的一点就是：所有的字符都是UTF-8编码的。接下来让我们更加深入的来学习Go语言的 `regexp` 包相关知识吧。

通过正则判断是否匹配

`regexp` 包中含有三个函数用来判断是否匹配，如果匹配返回true，否则返回false

```
func Match(pattern string, b []byte) (matched bool, error error)
func MatchReader(pattern string, r io.RuneReader) (matched bool, er
func MatchString(pattern string, s string) (matched bool, error err
```

上面的三个函数实现了同一个功能，就是判断 `pattern` 是否和输入源匹配，匹配的话就返回true，如果解析正则出错则返回error。三个函数的输入源分别是byte slice、RuneReader和string。

如果要验证一个输入是不是IP地址，那么如何来判断呢？请看如下实现

```
func IsIP(ip string) (b bool) {
    if m, _ := regexp.MatchString("^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]
        return false
    }
    return true
}
```

可以看到， `regexp` 的pattern和我们平常使用的正则一模一样。再来看一个例子：当用户输入一个字符串，我们想知道是不是一次合法的输入：

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Usage: regexp [string]")
        os.Exit(1)
    } else if m, _ := regexp.MatchString("^[0-9]+$", os.Args[1]); m
        fmt.Println("数字")
    } else {
        fmt.Println("不是数字")
    }
}
```

在上面的两个小例子中，我们采用了Match(Reader|String)来判断一些字符串是否符合我们的描述需求，它们使用起来非常方便。

通过正则获取内容

Match模式只能用来对字符串的判断，而无法截取字符串的一部分、过滤字符串、或者提取出符合条件的一批字符串。如果想要满足这些需求，那就需要使用正则表达式的复杂模式。

我们经常需要一些爬虫程序，下面就以爬虫为例来说明如何使用正则来过滤或截取抓取到的数据：

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "regexp"
    "strings"
)

func main() {
    resp, err := http.Get("http://www.baidu.com")
    if err != nil {
        fmt.Println("http get error.")
    }
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("http read error")
        return
    }

    src := string(body)

    //将HTML标签全转换成小写
    re, _ := regexp.Compile("\\<[\\S\\s]+?\\>")
    src = re.ReplaceAllStringFunc(src, strings.ToLower)

    //去除STYLE
    re, _ = regexp.Compile("\\<style[\\S\\s]+?\\</style\\>")
    src = re.ReplaceAllString(src, "")

    //去除SCRIPT
    re, _ = regexp.Compile("\\<script[\\S\\s]+?\\</script\\>")
    src = re.ReplaceAllString(src, "")

    //去除所有尖括号内的HTML代码，并换成换行符
    re, _ = regexp.Compile("\\<[\\S\\s]+?\\>")
    src = re.ReplaceAllString(src, "\n")

    //去除连续的换行符
    re, _ = regexp.Compile("\\s{2,}")
    src = re.ReplaceAllString(src, "\n")

    fmt.Println(strings.TrimSpace(src))
}
```

从这个示例可以看出，使用复杂的正则首先是Compile，它会解析正则表达式是否合法，如果正确，那么就会返回一个Regexp，然后就可以利用返回的Regexp在任意的字符串上面执行需要的操作。

解析正则表达式的有如下几个方法：

```
func Compile(expr string) (*Regexp, error)
func CompilePOSIX(expr string) (*Regexp, error)
func MustCompile(str string) *Regexp
func MustCompilePOSIX(str string) *Regexp
```

CompilePOSIX和Compile的不同点在于POSIX必须使用POSIX语法，它使用最左最长方式搜索，而Compile是采用的则只采用最左方式搜索(例如[a-z]{2,4}这样一个正则表达式，应用于"aa09aaa88aaaa"这个文本串时，CompilePOSIX返回了aaaa，而Compile的返回的是aa)。前缀有Must的函数表示，在解析正则语法的时候，如果匹配模式串不满足正确的语法则直接panic，而不加Must的则只是返回错误。

在了解了如何新建一个Regexp之后，我们再来看一下这个struct提供了哪些方法来辅助我们操作字符串，首先我们来看下面这些用来搜索的函数：

```
func (re *Regexp) Find(b []byte) []byte
func (re *Regexp) FindAll(b []byte, n int) [][]byte
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
func (re *Regexp) FindAllString(s string, n int) []string
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [][]i
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
func (re *Regexp) FindIndex(b []byte) (loc []int)
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int
func (re *Regexp) FindString(s string) string
func (re *Regexp) FindStringIndex(s string) (loc []int)
func (re *Regexp) FindStringSubmatch(s string) []string
func (re *Regexp) FindStringSubmatchIndex(s string) []int
func (re *Regexp) FindSubmatch(b []byte) [][]byte
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

上面这18个函数我们根据输入源(byte slice、string和io.RuneReader)不同还可以继续简化成如下几个，其他的只是输入源不一样，其他功能基本是一样的：

```
func (re *Regexp) Find(b []byte) []byte
func (re *Regexp) FindAll(b []byte, n int) [][]byte
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
func (re *Regexp) FindIndex(b []byte) (loc []int)
func (re *Regexp) FindSubmatch(b []byte) [][]byte
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

对于这些函数的使用我们来看下面这个例子

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    a := "I am learning Go language"

    re, _ := regexp.Compile("[a-z]{2,4}")

    //查找符合正则的第一个
    one := re.Find([]byte(a))
    fmt.Println("Find:", string(one))

    //查找符合正则的所有slice,n小于0表示返回全部符合的字符串，不然就是返回指定
    all := re.FindAll([]byte(a), -1)
    fmt.Println("FindAll", all)

    //查找符合条件的index位置,开始位置和结束位置
    index := re.FindIndex([]byte(a))
    fmt.Println("FindIndex", index)

    //查找符合条件的所有的index位置，n同上
    allindex := re.FindAllIndex([]byte(a), -1)
    fmt.Println("FindAllIndex", allindex)

    re2, _ := regexp.Compile("am(.*)lang(.*)")

    //查找Submatch,返回数组，第一个元素是匹配的全部元素，第二个元素是第一个()
    //下面的输出第一个元素是"am learning Go language"
    //第二个元素是" learning Go "，注意包含空格的输出
    //第三个元素是"uage"
    submatch := re2.FindSubmatch([]byte(a))
    fmt.Println("FindSubmatch", submatch)
    for _, v := range submatch {
        fmt.Println(string(v))
    }

    //定义和上面的FindIndex一样
    submatchindex := re2.FindSubmatchIndex([]byte(a))
    fmt.Println(submatchindex)

    //FindAllSubmatch,查找所有符合条件的子匹配
    submatchall := re2.FindAllSubmatch([]byte(a), -1)
    fmt.Println(submatchall)

    //FindAllSubmatchIndex,查找所有字匹配的index
    submatchallindex := re2.FindAllSubmatchIndex([]byte(a), -1)
```

```
    fmt.Println(submatchallindex)
}
```

前面介绍过匹配函数，Regexp也定义了三个函数，它们和同名的外部函数功能一模一样，其实外部函数就是调用了这Regexp的三个函数来实现的：

```
func (re *Regexp) Match(b []byte) bool
func (re *Regexp) MatchReader(r io.RuneReader) bool
func (re *Regexp) MatchString(s string) bool
```

接下里让我们来了解替换函数是怎么操作的？

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []by
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
func (re *Regexp) ReplaceAllString(src, repl string) string
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string
```

这些替换函数我们在上面的抓网页的例子有详细应用示例，

接下来我们看一下Expand的解释：

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, m
func (re *Regexp) ExpandString(dst []byte, template string, src str
```

那么这个Expand到底用来干嘛的呢？请看下面的例子：

```
func main() {
    src := []byte(`
        call hello alice
        hello bob
        call hello eve
    `)
    pat := regexp.MustCompile(`(?m)(call)\s+(?P<cmd>\w+)\s+(?P<arg>
    res := []byte{}
    for _, s := range pat.FindAllSubmatchIndex(src, -1) {
        res = pat.Expand(res, []byte("$cmd('$arg')\n"), src, s)
    }
    fmt.Println(string(res))
}
```

至此我们已经全部介绍完Go语言的 `regexp` 包，通过对它的主要函数介绍及演示，相信大家应该能够通过Go语言的正则包进行一些基本的正则的操作了。

links

- 目录
- 上一节: Json处理
- 下一节: 模板处理

7.4 模板处理

什么是模板

你一定听说过一种叫做MVC的设计模式，Model处理数据，View展现结果，Controller控制用户的请求，至于View层的处理，在很多动态语言里面都是通过在静态HTML中插入动态语言生成的数据，例如JSP中通过插入 `<%=....=%>` ，PHP中通过插入 `<?php.....?>` 来实现的。

通过下面这个图可以说明模板的机制

图7.1 模板机制图

Web应用反馈给客户端的信息中的大部分内容是静态的，不变的，而另外少部分是根据用户的请求来动态生成的，例如要显示用户的访问记录列表。用户之间只有记录数据是不同的，而列表的样式则是固定的，此时采用模板可以复用很多静态代码。

Go模板使用

在Go语言中，我们使用 `template` 包来进行模板处理，使用类似 `Parse` 、 `ParseFile` 、 `Execute` 等方法从文件或者字符串加载模板，然后执行类似上面图片展示的模板的merge操作。请看下面的例子：

```
func handler(w http.ResponseWriter, r *http.Request) {
    t := template.New("some template") //创建一个模板
    t, _ = t.ParseFiles("tmpl/welcome.html", nil)  //解析模板文件
    user := GetUser() //获取当前用户信息
    t.Execute(w, user)  //执行模板的merger操作
}
```

通过上面的例子我们可以看到Go语言的模板操作非常的简单方便，和其他语言的模板处理类似，都是先获取数据，然后渲染数据。

为了演示和测试代码的方便，我们在接下来的例子中采用如下格式的代码

- 使用Parse代替ParseFiles，因为Parse可以直接测试一个字符串，而不需要额外的文件
- 不使用handler来写演示代码，而是每个测试一个main，方便测试
- 使用 `os.Stdout` 代替 `http.ResponseWriter` ，因为 `os.Stdout` 实现了 `io.Writer` 接口

模板中如何插入数据？

上面我们演示了如何解析并渲染模板，接下来让我们来更加详细的了解如何把数据渲染出来。一个模板都是应用在一个Go的对象之上，Go对象的字段如何插入到模板中呢？

字段操作

Go语言的模板通过 `{{}}` 来包含需要在渲染时被替换的字段， `{{.}}` 表示当前的对象，这和Java或者C++中的this类似，如果要访问当前对象的字段通过 `{{.FieldName}}` ,但是需要注意一点：这个字段必须是导出的(字段首字母必须是大写的),否则在渲染的时候就会报错，请看下面的这个例子：

```
package main

import (
    "html/template"
    "os"
)

type Person struct {
    UserName string
}

func main() {
    t := template.New("fieldname example")
    t, _ = t.Parse("hello {{.UserName}}!")
    p := Person{UserName: "Astaxie"}
    t.Execute(os.Stdout, p)
}
```

上面的代码我们可以正确的输出 `hello Astaxie` ，但是如果我们稍微修改一下代码，在模板中含有了未导出的字段，那么就会报错

```
type Person struct {
    UserName string
    email    string  //未导出的字段，首字母是小写的
}

t, _ = t.Parse("hello {{.UserName}}! {{.email}}")
```

上面的代码就会报错，因为我们调用了一个未导出的字段，但是如果我们调用了一个不存在的字段是不会报错的，而是输出为空。

如果模板中输出 `{{.}}` ，这个一般应用于字符串对象，默认会调用fmt包输出字符串的内容。

输出嵌套字段内容

上面我们例子展示了如何针对一个对象的字段输出，那么如果字段里面还有对象，如何来循环的输出这些内容呢？我们可以使用 `{{with …}}…{{end}}` 和 `{{range …}}{{end}}` 来进行数据的输出。

- {{range}} 这个和Go语法里面的range类似，循环操作数据
- {{with}}操作是指当前对象的值，类似上下文的概念

详细的使用请看下面的例子：

```
package main

import (
    "html/template"
    "os"
)

type Friend struct {
    Fname string
}

type Person struct {
    UserName string
    Emails   []string
    Friends  []*Friend
}

func main() {
    f1 := Friend{Fname: "minux.ma"}
    f2 := Friend{Fname: "xushiwei"}
    t := template.New("fieldname example")
    t, _ = t.Parse(`hello {{.UserName}}!
            {{range .Emails}}
                an email {{.}}
            {{end}}
            {{with .Friends}}
            {{range .}}
                my friend name is {{.Fname}}
            {{end}}
            {{end}}
            `)
    p := Person{UserName: "Astaxie",
        Emails:  []string{"astaxie@beego.me", "astaxie@gmail.com"},
        Friends: []*Friend{&f1, &f2}}
    t.Execute(os.Stdout, p)
}
```

条件处理

在Go模板里面如果需要进行条件判断，那么我们可以使用和Go语言的 `if-else` 语法类似的方式来处理，如果pipeline为空，那么if就认为是false，下面的例子展示了如何使用 `if-else` 语法：

```
package main

import (
    "os"
    "text/template"
)

func main() {
    tEmpty := template.New("template test")
    tEmpty = template.Must(tEmpty.Parse("空 pipeline if demo: {{if 
    tEmpty.Execute(os.Stdout, nil)

    tWithValue := template.New("template test")
    tWithValue = template.Must(tWithValue.Parse("不为空的 pipeline i
    tWithValue.Execute(os.Stdout, nil)

    tIfElse := template.New("template test")
    tIfElse = template.Must(tIfElse.Parse("if-else demo: {{if `any
    tIfElse.Execute(os.Stdout, nil)
}
```

通过上面的演示代码我们知道 `if-else` 语法相当的简单，在使用过程中很容易集成到我们的模板代码中。

> 注意：if里面无法使用条件判断，例如.Mail=="astaxie@gmail.com"，这样的判断是不正确的，if里面只能是bool值

pipelines

Unix用户已经很熟悉什么是 `pipe` 了， `ls | grep "beego"` 类似这样的语法你是不是经常使用，过滤当前目录下面的文件，显示含有"beego"的数据，表达的意思就是前面的输出可以当做后面的输入，最后显示我们想要的数据，而Go语言模板最强大的一点就是支持pipe数据，在Go语言里面任何 `{{}}` 里面的都是pipelines数据，例如我们上面输出的email里面如果还有一些可能引起XSS注入的，那么我们如何来进行转化呢？

```
{{. | html}}
```

在email输出的地方我们可以采用如上方式可以把输出全部转化html的实体，上面的这种方式和我们平常写Unix的方式是不是一模一样，操作起来相当的简便，调用其他的函数也是类似的方式。

模板变量

有时候，我们在模板使用过程中需要定义一些局部变量，我们可以在一些操作中申明局部变量，例如 `with``range``if` 过程中申明局部变量，这个变量的作用域是 `{{end}}` 之前，Go语言通过申明的局部变量格式如下所示：

```
$variable := pipeline
```

详细的例子看下面的：

```
{{with $x := "output" | printf "%q"}}{{$x}}{{end}}
{{with $x := "output"}}{{printf "%q" $x}}{{end}}
{{with $x := "output"}}{{$x | printf "%q"}}{{end}}
```

模板函数

模板在输出对象的字段值时，采用了 `fmt` 包把对象转化成了字符串。但是有时候我们的需求可能不是这样的，例如有时候我们为了防止垃圾邮件发送者通过采集网页的方式来发送给我们的邮箱信息，我们希望把 `@` 替换成 `at` 例如： `astaxie at beego.me` ，如果要实现这样的功能，我们就需要自定义函数来做这个功能。

每一个模板函数都有一个唯一值的名字，然后与一个Go函数关联，通过如下的方式来关联

```
type FuncMap map[string]interface{}
```

例如，如果我们想要的email函数的模板函数名是 `emailDeal` ，它关联的Go函数名称是 `EmailDealWith` ,那么我们可以通过下面的方式来注册这个函数

```
t = t.Funcs(template.FuncMap{"emailDeal": EmailDealWith})
```

`EmailDealWith` 这个函数的参数和返回值定义如下：

```
func EmailDealWith(args …interface{}) string
```

我们来看下面的实现例子：

```
package main

import (
    "fmt"
```

```
    "html/template"
    "os"
    "strings"
)

type Friend struct {
    Fname string
}

type Person struct {
    UserName string
    Emails   []string
    Friends  []*Friend
}

func EmailDealWith(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    // find the @ symbol
    substrs := strings.Split(s, "@")
    if len(substrs) != 2 {
        return s
    }
    // replace the @ by " at "
    return (substrs[0] + " at " + substrs[1])
}

func main() {
    f1 := Friend{Fname: "minux.ma"}
    f2 := Friend{Fname: "xushiwei"}
    t := template.New("fieldname example")
    t = t.Funcs(template.FuncMap{"emailDeal": EmailDealWith})
    t, _ = t.Parse(`hello {{.UserName}}!
                {{range .Emails}}
                    an emails {{.|emailDeal}}
                {{end}}
                {{with .Friends}}
                {{range .}}
                    my friend name is {{.Fname}}
                {{end}}
                {{end}}
                `)
    p := Person{UserName: "Astaxie",
        Emails:  []string{"astaxie@beego.me", "astaxie@gmail.com"},
        Friends: []*Friend{&f1, &f2}}
    t.Execute(os.Stdout, p)
}
```

上面演示了如何自定义函数，其实，在模板包内部已经有内置的实现函数，下面代码截取自模板包里面

```
var builtins = FuncMap{
    "and":      and,
    "call":     call,
    "html":     HTMLEscaper,
    "index":    index,
    "js":       JSEscaper,
    "len":      length,
    "not":      not,
    "or":       or,
    "print":    fmt.Sprint,
    "printf":   fmt.Sprintf,
    "println":  fmt.Sprintln,
    "urlquery": URLQueryEscaper,
}
```

Must操作

模板包里面有一个函数 `Must`，它的作用是检测模板是否正确，例如大括号是否匹配，注释是否正确的关闭，变量是否正确的书写。接下来我们演示一个例子，用Must来判断模板是否正确：

```
package main

import (
    "fmt"
    "text/template"
)

func main() {
    tOk := template.New("first")
    template.Must(tOk.Parse(" some static text /* and a comment */'
    fmt.Println("The first one parsed OK.")

    template.Must(template.New("second").Parse("some static text {
    fmt.Println("The second one parsed OK.")

    fmt.Println("The next one ought to fail.")
    tErr := template.New("check parse error with Must")
    template.Must(tErr.Parse(" some static text {{ .Name }"))
}
```

讲输出如下内容

```
The first one parsed OK.
The second one parsed OK.
The next one ought to fail.
panic: template: check parse error with Must:1: unexpected "}" in c
```

嵌套模板

我们平常开发Web应用的时候，经常会遇到一些模板有些部分是固定不变的，然后可以抽取出来作为一个独立的部分，例如一个博客的头部和尾部是不变的，而唯一改变的是中间的内容部分。所以我们可以定义成 `header` 、 `content` 、 `footer` 三个部分。Go语言中通过如下的语法来申明

```
{{define "子模板名称"}}内容{{end}}
```

通过如下方式来调用：

```
{{template "子模板名称"}}
```

接下来我们演示如何使用嵌套模板，我们定义三个文件， `header.tmpl` 、 `content.tmpl` 、 `footer.tmpl` 文件，里面的内容如下

```
//header.tmpl
{{define "header"}}
<html>
<head>
    <title>演示信息</title>
</head>
<body>
{{end}}

//content.tmpl
{{define "content"}}
{{template "header"}}
<h1>演示嵌套</h1>
<ul>
    <li>嵌套使用define定义子模板</li>
    <li>调用使用template</li>
</ul>
{{template "footer"}}
{{end}}

//footer.tmpl
{{define "footer"}}
</body>
</html>
{{end}}
```

演示代码如下：

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    s1, _ := template.ParseFiles("header.tmpl", "content.tmpl", "fo
    s1.ExecuteTemplate(os.Stdout, "header", nil)
    fmt.Println()
    s1.ExecuteTemplate(os.Stdout, "content", nil)
    fmt.Println()
    s1.ExecuteTemplate(os.Stdout, "footer", nil)
    fmt.Println()
    s1.Execute(os.Stdout, nil)
}
```

通过上面的例子我们可以看到通过 `template.ParseFiles` 把所有的嵌套模板全部解析到模板里面，其实每一个定义的{{define}}都是一个独立的模板，他们相互独立，是并行存在的关系，内部其实存储的是类似map的一种关系(key是模板的名称，value是模板的内容)，然后我们通过 `ExecuteTemplate` 来执行相应的子模板内容，我们可以看到header、footer都是相对独立的，都能输出内容，content 中因为嵌套了header和footer的内容，就会同时输出三个的内容。但是当我们执行 `s1.Execute` ，没有任何的输出，因为在默认的情况下没有默认的子模板，所以不会输出任何的东西。

> 同一个集合类的模板是互相知晓的，如果同一模板被多个集合使用，则它需要在多个集合中分别解析

总结

通过上面对模板的详细介绍，我们了解了如何把动态数据与模板融合：如何输出循环数据、如何自定义函数、如何嵌套模板等等。通过模板技术的应用，我们可以完成MVC模式中V的处理，接下来的章节我们将介绍如何来处理M和C。

links

- 目录
- 上一节: 正则处理
- 下一节: 文件操作

7.5 文件操作

在任何计算机设备中，文件是都是必须的对象，而在Web编程中,文件的操作一直是Web程序员经常遇到的问题,文件操作在Web应用中是必须的,非常有用的,我们经常遇到生成文件目录,文件(夹)编辑等操作,现在我把Go中的这些操作做一详细总结并实例示范如何使用。

目录操作

文件操作的大多数函数都是在os包里面，下面列举了几个目录操作的：

- func Mkdir(name string, perm FileMode) error

  创建名称为name的目录，权限设置是perm，例如0777

- func MkdirAll(path string, perm FileMode) error

  根据path创建多级子目录，例如astaxie/test1/test2。

- func Remove(name string) error

  删除名称为name的目录，当目录下有文件或者其他目录是会出错

- func RemoveAll(path string) error

  根据path删除多级子目录，如果path是单个名称，那么该目录下的子目录全部删除。

下面是演示代码：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    os.Mkdir("astaxie", 0777)
    os.MkdirAll("astaxie/test1/test2", 0777)
    err := os.Remove("astaxie")
    if err != nil {
        fmt.Println(err)
    }
    os.RemoveAll("astaxie")
}
```

文件操作

建立与打开文件

新建文件可以通过如下两个方法

- func Create(name string) (file *File, err Error)

  根据提供的文件名创建新的文件，返回一个文件对象，默认权限是0666的文件，返回的文件对象是可读写的。

- func NewFile(fd uintptr, name string) *File

  根据文件描述符创建相应的文件，返回一个文件对象

通过如下两个方法来打开文件：

- func Open(name string) (file *File, err Error)

  该方法打开一个名称为name的文件，但是是只读方式，内部实现其实调用了OpenFile。

- func OpenFile(name string, flag int, perm uint32) (file *File, err Error)

  打开名称为name的文件，flag是打开的方式，只读、读写等，perm是权限

写文件

写文件函数：

- func (file *File) Write(b []byte) (n int, err Error)

  写入byte类型的信息到文件

- func (file *File) WriteAt(b []byte, off int64) (n int, err Error)

  在指定位置开始写入byte类型的信息

- func (file *File) WriteString(s string) (ret int, err Error)

  写入string信息到文件

写文件的示例代码

```
package main

import (
    "fmt"
    "os"
)

func main() {
    userFile := "astaxie.txt"
    fout, err := os.Create(userFile)
    if err != nil {
        fmt.Println(userFile, err)
        return
    }
    defer fout.Close()
    for i := 0; i < 10; i++ {
        fout.WriteString("Just a test!\r\n")
        fout.Write([]byte("Just a test!\r\n"))
    }
}
```

读文件

读文件函数：

- func (file *File) Read(b []byte) (n int, err Error)

  读取数据到b中

- func (file *File) ReadAt(b []byte, off int64) (n int, err Error)

  从off开始读取数据到b中

读文件的示例代码:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    userFile := "asatxie.txt"
    fl, err := os.Open(userFile)
    if err != nil {
        fmt.Println(userFile, err)
        return
    }
    defer fl.Close()
    buf := make([]byte, 1024)
    for {
        n, _ := fl.Read(buf)
        if 0 == n {
            break
        }
        os.Stdout.Write(buf[:n])
    }
}
```

删除文件

Go语言里面删除文件和删除文件夹是同一个函数

- func Remove(name string) Error

  调用该函数就可以删除文件名为name的文件

links

- 目录
- 上一节: 模板处理
- 下一节: 字符串处理

7.6 字符串处理

字符串在我们平常的Web开发中经常用到，包括用户的输入，数据库读取的数据等，我们经常需要对字符串进行分割、连接、转换等操作，本小节将通过Go标准库中的strings和strconv两个包中的函数来讲解如何进行有效快速的操作。

字符串操作

下面这些函数来自于strings包，这里介绍一些我平常经常用到的函数，更详细的请参考官方的文档。

- func Contains(s, substr string) bool

  字符串s中是否包含substr，返回bool值

```
fmt.Println(strings.Contains("seafood", "foo"))
fmt.Println(strings.Contains("seafood", "bar"))
fmt.Println(strings.Contains("seafood", ""))
fmt.Println(strings.Contains("", ""))
//Output:
//true
//false
//true
//true
```

- func Join(a []string, sep string) string

  字符串链接，把slice a通过sep链接起来

```
s := []string{"foo", "bar", "baz"}
fmt.Println(strings.Join(s, ", "))
//Output:foo, bar, baz
```

- func Index(s, sep string) int

  在字符串s中查找sep所在的位置，返回位置值，找不到返回-1

```
fmt.Println(strings.Index("chicken", "ken"))
fmt.Println(strings.Index("chicken", "dmr"))
//Output:4
//-1
```

- func Repeat(s string, count int) string

重复s字符串count次，最后返回重复的字符串

```
fmt.Println("ba" + strings.Repeat("na", 2))
//Output:banana
```

- func Replace(s, old, new string, n int) string

  在s字符串中，把old字符串替换为new字符串，n表示替换的次数，小于0表示全部替换

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))
fmt.Println(strings.Replace("oink oink oink", "oink", "moo",
//Output:oinky oinky oink
//moo moo moo
```

- func Split(s, sep string) []string

  把s字符串按照sep分割，返回slice

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
fmt.Printf("%q\n", strings.Split("a man a plan a canal panama
fmt.Printf("%q\n", strings.Split(" xyz ", ""))
fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
//Output:["a" "b" "c"]
//["" "man " "plan " "canal panama"]
//[" " "x" "y" "z" " "]
//[""]
```

- func Trim(s string, cutset string) string

  在s字符串的头部和尾部去除cutset指定的字符串

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung !!! ", "! "))
//Output:["Achtung"]
```

- func Fields(s string) []string

  去除s字符串的空格符，并且按照空格分割返回slice

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz
//Output:Fields are: ["foo" "bar" "baz"]
```

字符串转换

字符串转化的函数在strconv中，如下也只是列出一些常用的：

- Append 系列函数将整数等转换为字符串后，添加到现有的字节数组中。

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    str := make([]byte, 0, 100)
    str = strconv.AppendInt(str, 4567, 10)
    str = strconv.AppendBool(str, false)
    str = strconv.AppendQuote(str, "abcdefg")
    str = strconv.AppendQuoteRune(str, '单')
    fmt.Println(string(str))
}
```

- Format 系列函数把其他类型的转换为字符串

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    a := strconv.FormatBool(false)
    b := strconv.FormatFloat(123.23, 'g', 12, 64)
    c := strconv.FormatInt(1234, 10)
    d := strconv.FormatUint(12345, 10)
    e := strconv.Itoa(1023)
    fmt.Println(a, b, c, d, e)
}
```

- Parse 系列函数把字符串转换为其他类型

```
package main

import (
    "fmt"
    "strconv"
)
func checkError(e error){
    if e != nil{
        fmt.Println(e)
    }
}
func main() {
    a, err := strconv.ParseBool("false")
    checkError(err)
    b, err := strconv.ParseFloat("123.23", 64)
    checkError(err)
    c, err := strconv.ParseInt("1234", 10, 64)
    checkError(err)
    d, err := strconv.ParseUint("12345", 10, 64)
    checkError(err)
    e, err := strconv.Atoi("1023")
    checkError(err)
    fmt.Println(a, b, c, d, e)
}
```

links

- 目录
- 上一节: 文件操作
- 下一节: 小结

7.7 小结

这一章给大家介绍了一些文本处理的工具，包括XML、JSON、正则和模板技术，XML和JSON是数据交互的工具，通过XML和JSON你可以表达各种含义，通过正则你可以处理文本(搜索、替换、截取)，通过模板技术你可以展现这些数据给用户。这些都是你开发Web应用过程中需要用到的技术，通过这个小节的介绍你能够了解如何处理文本、展现文本。

links

- 目录
- 上一节: 字符串处理
- 下一节: Web服务

8 Web服务

Web服务可以让你在HTTP协议的基础上通过XML或者JSON来交换信息。如果你想知道上海的天气预报、中国石油的股价或者淘宝商家的一个商品信息，你可以编写一段简短的代码，通过抓取这些信息然后通过标准的接口开放出来，就如同你调用一个本地函数并返回一个值。

Web服务背后的关键在于平台的无关性，你可以运行你的服务在Linux系统，可以与其他Windows的asp.net程序交互，同样的，也可以通过同一个接口和运行在FreeBSD上面的JSP无障碍地通信。

目前主流的有如下几种Web服务：REST、SOAP。

REST请求是很直观的，因为REST是基于HTTP协议的一个补充，他的每一次请求都是一个HTTP请求，然后根据不同的method来处理不同的逻辑，很多Web开发者都熟悉HTTP协议，所以学习REST是一件比较容易的事情。所以我们在8.3小节将详细的讲解如何在Go语言中来实现REST方式。

SOAP是W3C在跨网络信息传递和远程计算机函数调用方面的一个标准。但是SOAP非常复杂，其完整的规范篇幅很长，而且内容仍然在增加。Go语言是以简单著称，所以我们不会介绍SOAP这样复杂的东西。而Go语言提供了一种天生性能很不错，开发起来很方便的RPC机制，我们将会在8.4小节详细介绍如何使用Go语言来实现RPC。

Go语言是21世纪的C语言，我们追求的是性能、简单，所以我们在8.1小节里面介绍如何使用Socket编程，很多游戏服务都是采用Socket来编写服务端，因为HTTP协议相对而言比较耗费性能，让我们看看Go语言如何来Socket编程。目前随着HTML5的发展，webSockets也逐渐的成为很多页游公司接下来开发的一些手段，我们将在8.2小节里面讲解Go语言如何编写webSockets的代码。

目录

links

- 目录
- 上一章: 第七章总结
- 下一节: Socket编程

8.1 Socket编程

在很多底层网络应用开发者的眼里一切编程都是Socket，话虽然有点夸张，但却也几乎如此了，现在的网络编程几乎都是用Socket来编程。你想过这些情景么？我们每天打开浏览器浏览网页时，浏览器进程怎么和Web服务器进行通信的呢？当你用QQ聊天时，QQ进程怎么和服务器或者是你的好友所在的QQ进程进行通信的呢？当你打开PPstream观看视频时，PPstream进程如何与视频服务器进行通信的呢？如此种种，都是靠Socket来进行通信的，以一斑窥全豹，可见Socket编程在现代编程中占据了多么重要的地位，这一节我们将介绍Go语言中如何进行Socket编程。

什么是Socket？

Socket起源于Unix，而Unix基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。Socket就是该模式的一个实现，网络的Socket数据传输是一种特殊的I/O，Socket也是一种文件描述符。Socket也具有一个类似于打开文件的函数调用：Socket()，该函数返回一个整型的Socket描述符，随后的连接建立、数据传输等操作都是通过该Socket实现的。

常用的Socket类型有两种：流式Socket（SOCK_STREAM）和数据报式Socket（SOCK_DGRAM）。流式是一种面向连接的Socket，针对于面向连接的TCP服务应用；数据报式Socket是一种无连接的Socket，对应于无连接的UDP服务应用。

Socket如何通信

网络中的进程之间如何通过Socket通信呢？首要解决的问题是如何唯一标识一个进程，否则通信无从谈起！在本地可以通过进程PID来唯一标识一个进程，但是在网络中这是行不通的。其实TCP/IP协议族已经帮我们解决了这个问题，网络层的“ip地址”可以唯一标识网络中的主机，而传输层的“协议+端口”可以唯一标识主机中的应用程序（进程）。这样利用三元组（ip地址，协议，端口）就可以标识网络的进程了，网络中需要互相通信的进程，就可以利用这个标志在他们之间进行交互。请看下面这个TCP/IP协议结构图

图8.1 七层网络协议图

使用TCP/IP协议的应用程序通常采用应用编程接口：UNIX BSD的套接字（socket）和UNIX System V的TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用socket，而现在又是网络时代，网络中进程通信是无处不在，这就是为什么说“一切皆Socket”。

Socket基础知识

通过上面的介绍我们知道Socket有两种：TCP Socket和UDP Socket，TCP和UDP是协议，而要确定一个进程的需要三元组，需要IP地址和端口。

IPv4地址

目前的全球因特网所采用的协议族是TCP/IP协议。IP是TCP/IP协议中网络层的协议，是TCP/IP协议族的核心协议。目前主要采用的IP协议的版本号是4(简称为IPv4)，发展至今已经使用了30多年。

IPv4的地址位数为32位，也就是最多有2的32次方的网络设备可以联到Internet上。近十年来由于互联网的蓬勃发展，IP位址的需求量愈来愈大，使得IP位址的发放愈趋紧张，前一段时间，据报道IPV4的地址已经发放完毕，我们公司目前很多服务器的IP都是一个宝贵的资源。

地址格式类似这样：127.0.0.1 172.122.121.111

IPv6地址

IPv6是下一版本的互联网协议，也可以说是下一代互联网的协议，它是为了解决IPv4在实施过程中遇到的各种问题而被提出的，IPv6采用128位地址长度，几乎可以不受限制地提供地址。按保守方法估算IPv6实际可分配的地址，整个地球的每平方米面积上仍可分配1000多个地址。在IPv6的设计过程中除了一劳永逸地解决了地址短缺问题以外，还考虑了在IPv4中解决不好的其它问题，主要有端到端IP连接、服务质量（QoS）、安全性、多播、移动性、即插即用等。

地址格式类似这样：2002:c0e8:82e7:0:0:0:c0e8:82e7

Go支持的IP类型

在Go的 `net` 包中定义了很多类型、函数和方法用来网络编程，其中IP的定义如下：

```
type IP []byte
```

在 `net` 包中有很多函数来操作IP，但是其中比较有用的也就几个，其中 `ParseIP(s string) IP` 函数会把一个IPv4或者IPv6的地址转化成IP类型，请看下面的例子:

```
package main
import (
    "net"
    "os"
    "fmt"
)
func main() {
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "Usage: %s ip-addr\n", os.Args[0])
        os.Exit(1)
    }
    name := os.Args[1]
    addr := net.ParseIP(name)
    if addr == nil {
        fmt.Println("Invalid address")
    } else {
        fmt.Println("The address is ", addr.String())
    }
    os.Exit(0)
}
```

执行之后你就会发现只要你输入一个IP地址就会给出相应的IP格式

TCP Socket

当我们知道如何通过网络端口访问一个服务时，那么我们能够做什么呢？作为客户端来说，我们可以通过向远端某台机器的的某个网络端口发送一个请求，然后得到在机器的此端口上监听的服务反馈的信息。作为服务端，我们需要把服务绑定到某个指定端口，并且在此端口上监听，当有客户端来访问时能够读取信息并且写入反馈信息。

在Go语言的 `net` 包中有一个类型 `TCPConn` ，这个类型可以用来作为客户端和服务器端交互的通道，他有两个主要的函数：

```
func (c *TCPConn) Write(b []byte) (n int, err os.Error)
func (c *TCPConn) Read(b []byte) (n int, err os.Error)
```

`TCPConn` 可以用在客户端和服务器端来读写数据。

还有我们需要知道一个 `TCPAddr` 类型，他表示一个TCP的地址信息，他的定义如下：

```
type TCPAddr struct {
    IP IP
    Port int
}
```

在Go语言中通过 `ResolveTCPAddr` 获取一个 `TCPAddr`

```
func ResolveTCPAddr(net, addr string) (*TCPAddr, os.Error)
```

- net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCP(IPv4-only),TCP(IPv6-only)或者TCP(IPv4,IPv6的任意一个).
- addr表示域名或者IP地址，例如"www.google.com:80" 或者"127.0.0.1:22".

TCP client

Go语言中通过net包中的 `DialTCP` 函数来建立一个TCP连接，并返回一个 `TCPConn` 类型的对象，当连接建立时服务器端也创建一个同类型的对象，此时客户端和服务器段通过各自拥有的 `TCPConn` 对象来进行数据交换。一般而言，客户端通过 `TCPConn` 对象将请求信息发送到服务器端，读取服务器端响应的信息。服务器端读取并解析来自客户端的请求，并返回应答信息，这个连接只有当任一端关闭了连接之后才失效，不然这连接可以一直在使用。建立连接的函数定义如下：

```
func DialTCP(net string, laddr, raddr *TCPAddr) (c *TCPConn, err os
```

- net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCP(IPv4-only)、TCP(IPv6-only)或者TCP(IPv4,IPv6的任意一个)
- laddr表示本机地址，一般设置为nil
- raddr表示远程的服务地址

接下来我们写一个简单的例子，模拟一个基于HTTP协议的客户端请求去连接一个Web服务端。我们要写一个简单的http请求头，格式类似如下：

```
"HEAD / HTTP/1.0\r\n\r\n"
```

从服务端接收到的响应信息格式可能如下：

```
HTTP/1.0 200 OK
ETag: "-9985996"
Last-Modified: Thu, 25 Mar 2010 17:51:10 GMT
Content-Length: 18074
Connection: close
Date: Sat, 28 Aug 2010 00:43:48 GMT
Server: lighttpd/1.4.23
```

我们的客户端代码如下所示：

```
package main

import (
    "fmt"
    "io/ioutil"
    "net"
    "os"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "Usage: %s host:port ", os.Args[0])
        os.Exit(1)
    }
    service := os.Args[1]
    tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
    checkError(err)
    conn, err := net.DialTCP("tcp", nil, tcpAddr)
    checkError(err)
    _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
    checkError(err)
    result, err := ioutil.ReadAll(conn)
    checkError(err)
    fmt.Println(string(result))
    os.Exit(0)
}
func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
        os.Exit(1)
    }
}
```

通过上面的代码我们可以看出：首先程序将用户的输入作为参数 `service` 传入 `net.ResolveTCPAddr` 获取一个tcpAddr,然后把tcpAddr传入DialTCP后创建了一个TCP连接 `conn` ，通过 `conn` 来发送请求信息，最后通过 `ioutil.ReadAll` 从 `conn` 中读取全部的文本，也就是服务端响应反馈的信息。

TCP server

上面我们编写了一个TCP的客户端程序，也可以通过net包来创建一个服务器端程序，在服务器端我们需要绑定服务到指定的非激活端口，并监听此端口，当有客户端请求到达的时候可以接收到来自客户端连接的请求。net包中有相应功能的函数，函数定义如下：

```
func ListenTCP(net string, laddr *TCPAddr) (l *TCPListener, err os.
func (l *TCPListener) Accept() (c Conn, err os.Error)
```

参数说明同DialTCP的参数一样。下面我们实现一个简单的时间同步服务，监听7777端口

```
package main

import (
    "fmt"
    "net"
    "os"
    "time"
)

func main() {
    service := ":7777"
    tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
    checkError(err)
    listener, err := net.ListenTCP("tcp", tcpAddr)
    checkError(err)
    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        daytime := time.Now().String()
        conn.Write([]byte(daytime)) // don't care about return valu
        conn.Close()                // we're finished with this cli
    }
}
func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
        os.Exit(1)
    }
}
```

上面的服务跑起来之后，它将会一直在那里等待，直到有新的客户端请求到达。当有新的客户端请求到达并同意接受 `Accept` 该请求的时候他会反馈当前的时间信息。值得注意的是，在代码中 `for` 循环里，当有错误发生时，直接continue而不是退出，是因为在服务器端跑代码的时候，当有错误发生的情况下最好是由服务端记录错误，然后当前连接的客户端直接报错而退出，从而不会影响到当前服务端运行的整个服务。

上面的代码有个缺点，执行的时候是单任务的，不能同时接收多个请求，那么该如何改造以使它支持多并发呢？Go里面有一个goroutine机制，请看下面改造后的代码

```
package main

import (
    "fmt"
    "net"
    "os"
    "time"
)

func main() {
    service := ":1200"
    tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
    checkError(err)
    listener, err := net.ListenTCP("tcp", tcpAddr)
    checkError(err)
    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        go handleClient(conn)
    }
}

func handleClient(conn net.Conn) {
    defer conn.Close()
    daytime := time.Now().String()
    conn.Write([]byte(daytime)) // don't care about return value
    // we're finished with this client
}
func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
        os.Exit(1)
    }
}
```

通过把业务处理分离到函数 `handleClient` ，我们就可以进一步地实现多并发执行了。看上去是不是很帅，增加 `go` 关键词就实现了服务端的多并发，从这个小例子也可以看出goroutine的强大之处。

有的朋友可能要问：这个服务端没有处理客户端实际请求的内容。如果我们需要通过从客户端发送不同的请求来获取不同的时间格式，而且需要一个长连接，该怎么做呢？请看：

```
package main

import (
    "fmt"
    "net"
    "os"
    "time"
    "strconv"
    "strings"
)

func main() {
    service := ":1200"
    tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
    checkError(err)
    listener, err := net.ListenTCP("tcp", tcpAddr)
    checkError(err)
    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        go handleClient(conn)
    }
}

func handleClient(conn net.Conn) {
    conn.SetReadDeadline(time.Now().Add(2 * time.Minute)) // set 2
    request := make([]byte, 128) // set maxium request length to 12
    defer conn.Close()  // close connection before exit
    for {
        read_len, err := conn.Read(request)

        if err != nil {
            fmt.Println(err)
            break
        }

        if read_len == 0 {
            break // connection already closed by client
        } else if strings.TrimSpace(string(request[:read_len])) ==
            daytime := strconv.FormatInt(time.Now().Unix(), 10)
            conn.Write([]byte(daytime))
        } else {
```

```
                daytime := time.Now().String()
                conn.Write([]byte(daytime))
            }

            request = make([]byte, 128) // clear last read content
        }
}

func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
        os.Exit(1)
    }
}
```

在上面这个例子中，我们使用 `conn.Read()` 不断读取客户端发来的请求。由于我们需要保持与客户端的长连接，所以不能在读取完一次请求后就关闭连接。由于 `conn.SetReadDeadline()` 设置了超时，当一定时间内客户端无请求发送，`conn` 便会自动关闭，下面的for循环即会因为连接已关闭而跳出。需要注意的是，`request` 在创建时需要指定一个最大长度以防止flood attack；每次读取到请求处理完毕后，需要清理request，因为 `conn.Read()` 会将新读取到的内容append到原内容之后。

控制TCP连接

TCP有很多连接控制函数，我们平常用到比较多的有如下几个函数：

```
func DialTimeout(net, addr string, timeout time.Duration) (Conn, er
```

设置建立连接的超时时间，客户端和服务器端都适用，当超过设置时间时，连接自动关闭。

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

用来设置写入/读取一个连接的超时时间。当超过设置时间时，连接自动关闭。

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

设置客户端是否和服务器端保持长连接，可以降低建立TCP连接时的握手开销，对于一些需要频繁交换数据的应用场景比较适用。

更多的内容请查看 `net` 包的文档。

UDP Socket

Go语言包中处理UDP Socket和TCP Socket不同的地方就是在服务器端处理多个客户端请求数据包的方式不同,UDP缺少了对客户端连接请求的Accept函数。其他基本几乎一模一样，只有TCP换成了UDP而已。UDP的几个主要函数如下所示：

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, os.Error)
func DialUDP(net string, laddr, raddr *UDPAddr) (c *UDPConn, err os
func ListenUDP(net string, laddr *UDPAddr) (c *UDPConn, err os.Erro
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err 
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (n int, err o
```

一个UDP的客户端代码如下所示,我们可以看到不同的就是TCP换成了UDP而已：

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
        os.Exit(1)
    }
    service := os.Args[1]
    udpAddr, err := net.ResolveUDPAddr("udp4", service)
    checkError(err)
    conn, err := net.DialUDP("udp", nil, udpAddr)
    checkError(err)
    _, err = conn.Write([]byte("anything"))
    checkError(err)
    var buf [512]byte
    n, err := conn.Read(buf[0:])
    checkError(err)
    fmt.Println(string(buf[0:n]))
    os.Exit(0)
}
func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
        os.Exit(1)
    }
}
```

我们来看一下UDP服务器端如何来处理：

```
package main

import (
    "fmt"
    "net"
    "os"
    "time"
)

func main() {
    service := ":1200"
    udpAddr, err := net.ResolveUDPAddr("udp4", service)
    checkError(err)
    conn, err := net.ListenUDP("udp", udpAddr)
    checkError(err)
    for {
        handleClient(conn)
    }
}
func handleClient(conn *net.UDPConn) {
    var buf [512]byte
    _, addr, err := conn.ReadFromUDP(buf[0:])
    if err != nil {
        return
    }
    daytime := time.Now().String()
    conn.WriteToUDP([]byte(daytime), addr)
}
func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
        os.Exit(1)
    }
}
```

总结

通过对TCP和UDP Socket编程的描述和实现，可见Go已经完备地支持了Socket编程，而且使用起来相当的方便，Go提供了很多函数，通过这些函数可以很容易就编写出高性能的Socket应用。

links

- 目录
- 上一节: Web服务
- 下一节: WebSocket

8.2 WebSocket

WebSocket是HTML5的重要特性，它实现了基于浏览器的远程socket，它使浏览器和服务器可以进行全双工通信，许多浏览器（Firefox、Google Chrome和Safari）都已对此做了支持。

在WebSocket出现之前，为了实现即时通信，采用的技术都是“轮询”，即在特定的时间间隔内，由浏览器对服务器发出HTTP Request，服务器在收到请求后，返回最新的数据给浏览器刷新，“轮询”使得浏览器需要对服务器不断发出请求，这样会占用大量带宽。

WebSocket采用了一些特殊的报头，使得浏览器和服务器只需要做一个握手的动作，就可以在浏览器和服务器之间建立一条连接通道。且此连接会保持在活动状态，你可以使用JavaScript来向连接写入或从中接收数据，就像在使用一个常规的TCP Socket一样。它解决了Web实时化的问题，相比传统HTTP有如下好处：

- 一个Web客户端只建立一个TCP连接
- Websocket服务端可以推送(push)数据到web客户端.
- 有更加轻量级的头，减少数据传送量

WebSocket URL的起始输入是ws://或是wss://（在SSL上）。下图展示了WebSocket的通信过程，一个带有特定报头的HTTP握手被发送到了服务器端，接着在服务器端或是客户端就可以通过JavaScript来使用某种套接口（socket），这一套接口可被用来通过事件句柄异步地接收数据。

图8.2 WebSocket原理图

WebSocket原理

WebSocket的协议颇为简单，在第一次handshake通过以后，连接便建立成功，其后的通讯数据都是以”\x00″开头，以”\xFF”结尾。在客户端，这个是透明的，WebSocket组件会自动将原始数据“掐头去尾”。

浏览器发出WebSocket连接请求，然后服务器发出回应，然后连接建立成功，这个过程通常称为“握手” (handshaking)。请看下面的请求和反馈信息：

图8.3 WebSocket的request和response信息

在请求中的"Sec-WebSocket-Key"是随机的，对于整天跟编码打交到的程序员，一眼就可以看出来：这个是一个经过base64编码后的数据。服务器端接收到这个请求之后需要把这个字符串连接上一个固定的字符串：

```
258EAFA5-E914-47DA-95CA-C5AB0DC85B11
```

即：`f7cb4ezEAl6C3wRaU6JORA==`连接上那一串固定字符串，生成一个这样的字符串：

```
f7cb4ezEAl6C3wRaU6JORA==258EAFA5-E914-47DA-95CA-C5AB0DC85B11
```

对该字符串先用 sha1安全散列算法计算出二进制的值，然后用base64对其进行编码，即可以得到握手后的字符串：

```
rE91AJhfC+6JdVcVXOGJEADEJdQ=
```

将之作为响应头`Sec-WebSocket-Accept`的值反馈给客户端。

Go实现WebSocket

Go语言标准包里面没有提供对WebSocket的支持，但是在由官方维护的go.net子包中有对这个的支持，你可以通过如下的命令获取该包：

```
go get code.google.com/p/go.net/websocket
```

WebSocket分为客户端和服务端，接下来我们将实现一个简单的例子:用户输入信息，客户端通过WebSocket将信息发送给服务器端，服务器端收到信息之后主动Push信息到客户端，然后客户端将输出其收到的信息，客户端的代码如下：

```
<html>
<head></head>
<body>
    <script type="text/javascript">
        var sock = null;
        var wsuri = "ws://127.0.0.1:1234";

        window.onload = function() {

            console.log("onload");

            sock = new WebSocket(wsuri);

            sock.onopen = function() {
                console.log("connected to " + wsuri);
            }

            sock.onclose = function(e) {
                console.log("connection closed (" + e.code + ")");
            }

            sock.onmessage = function(e) {
                console.log("message received: " + e.data);
            }
        };

        function send() {
            var msg = document.getElementById('message').value;
            sock.send(msg);
        };
    </script>
    <h1>WebSocket Echo Test</h1>
    <form>
        <p>
            Message: <input id="message" type="text" value="Hello,
        </p>
    </form>
    <button onclick="send();">Send Message</button>
</body>
</html>
```

可以看到客户端JS，很容易的就通过WebSocket函数建立了一个与服务器的连接sock，当握手成功后，会触发WebScoket对象的onopen事件，告诉客户端连接已经成功建立。客户端一共绑定了四个事件。

- 1）onopen 建立连接后触发
- 2）onmessage 收到消息后触发
- 3）onerror 发生错误时触发
- 4）onclose 关闭连接时触发

我们服务器端的实现如下：

```
package main

import (
    "code.google.com/p/go.net/websocket"
    "fmt"
    "log"
    "net/http"
)

func Echo(ws *websocket.Conn) {
    var err error

    for {
        var reply string

        if err = websocket.Message.Receive(ws, &reply); err != nil
            fmt.Println("Can't receive")
            break
        }

        fmt.Println("Received back from client: " + reply)

        msg := "Received:  " + reply
        fmt.Println("Sending to client: " + msg)

        if err = websocket.Message.Send(ws, msg); err != nil {
            fmt.Println("Can't send")
            break
        }
    }
}

func main() {
    http.Handle("/", websocket.Handler(Echo))

    if err := http.ListenAndServe(":1234", nil); err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}
```

当客户端将用户输入的信息Send之后，服务器端通过Receive接收到了相应信息，然后通过Send发送了应答信息。

图8.4 WebSocket服务器端接收到的信息

通过上面的例子我们看到客户端和服务器端实现WebSocket非常的方便，Go的源码net分支中已经实现了这个的协议，我们可以直接拿来用，目前随着HTML5的发展，我想未来WebSocket会是Web开发的一个重点，我们需要储备这方面的知识。

links

- 目录
- 上一节: Socket编程
- 下一节: REST

8.3 REST

RESTful，是目前最为流行的一种互联网软件架构。因为它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。本小节我们将来学习它到底是一种什么样的架构？以及在Go里面如何来实现它。

什么是REST

REST(REpresentational State Transfer)这个概念，首次出现是在 2000年Roy Thomas Fielding（他是HTTP规范的主要编写者之一）的博士论文中，它指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful的。

要理解什么是REST，我们需要理解下面几个概念:

- 资源（Resources） REST是"表现层状态转化"，其实它省略了主语。"表现层"其实指的是"资源"的"表现层"。

  那么什么是资源呢？就是我们平常上网访问的一张图片、一个文档、一个视频等。这些资源我们通过URI来定位，也就是一个URI表示一个资源。

- 表现层（Representation）

  资源是做一个具体的实体信息，他可以有多种的展现方式。而把实体展现出来就是表现层，例如一个txt文本信息，他可以输出成html、json、xml等格式，一个图片他可以jpg、png等方式展现，这个就是表现层的意思。

  URI确定一个资源，但是如何确定它的具体表现形式呢？应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

- 状态转化（State Transfer）

  访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，肯定涉及到数据和状态的变化。而HTTP协议是无状态的，那么这些状态肯定保存在服务器端，所以如果客户端想要通知服务器端改变数据和状态的变化，肯定要通过某种方式来通知它。

  客户端能通知服务器端的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。

综合上面的解释，我们总结一下什么是RESTful架构：

- （1）每一个URI代表一种资源；
- （2）客户端和服务器之间，传递这种资源的某种表现层；
- （3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状

态转化"。

Web应用要满足REST最重要的原则是:客户端和服务器之间的交互在请求之间是无状态的,即从客户端到服务器的每个请求都必须包含理解请求所必需的信息。如果服务器在请求之间的任何时间点重启，客户端不会得到通知。此外此请求可以由任何可用服务器回答，这十分适合云计算之类的环境。因为是无状态的，所以客户端可以缓存数据以改进性能。

另一个重要的REST原则是系统分层，这表示组件无法了解除了与它直接交互的层次以外的组件。通过将系统知识限制在单个层，可以限制整个系统的复杂性，从而促进了底层的独立性。

下图即是REST的架构图：

图8.5 REST架构图

当REST架构的约束条件作为一个整体应用时，将生成一个可以扩展到大量客户端的应用程序。它还降低了客户端和服务器之间的交互延迟。统一界面简化了整个系统架构，改进了子系统之间交互的可见性。REST简化了客户端和服务器的实现，而且对于使用REST开发的应用程序更加容易扩展。

下图展示了REST的扩展性：

图8.6 REST的扩展性

RESTful的实现

Go没有为REST提供直接支持，但是因为RESTful是基于HTTP协议实现的，所以我们可以利用 `net/http` 包来自己实现，当然需要针对REST做一些改造，REST是根据不同的method来处理相应的资源，目前已经存在的很多自称是REST的应用，其实并没有真正的实现REST，我暂且把这些应用根据实现的method分成几个级别，请看下图：

图8.7 REST的level分级

上图展示了我们目前实现REST的三个level，我们在应用开发的时候也不一定全部按照RESTful的规则全部实现他的方式，因为有些时候完全按照RESTful的方式未必是可行的，RESTful服务充分利用每一个HTTP方法，包括 `DELETE` 和 `PUT` 。可有时，HTTP客户端只能发出 `GET` 和 `POST` 请求：

- HTML标准只能通过链接和表单支持 `GET` 和 `POST` 。在没有Ajax支持的网页浏览器中不能发出 `PUT` 或 `DELETE` 命令

- 有些防火墙会挡住HTTP `PUT` 和 `DELETE` 请求要绕过这个限制，客户端需要把实际的 `PUT` 和 `DELETE` 请求通过 POST 请求穿透过来。RESTful 服务则要负责在收到的 POST 请求中找到原始的 HTTP 方法并还原。

我们现在可以通过 `POST` 里面增加隐藏字段 `_method` 这种方式可以来模拟 `PUT` 、 `DELETE` 等方式，但是服务器端需要做转换。我现在的项目里面就按照这种方式来做的REST接口。当然Go语言里面完全按照RESTful来实现是很容易的，我们通过下面的例子来说明如何实现RESTful的应用设计。

```
package main

import (
    "fmt"
    "github.com/drone/routes"
    "net/http"
)

func getuser(w http.ResponseWriter, r *http.Request) {
    params := r.URL.Query()
    uid := params.Get(":uid")
    fmt.Fprintf(w, "you are get user %s", uid)
}

func modifyuser(w http.ResponseWriter, r *http.Request) {
    params := r.URL.Query()
    uid := params.Get(":uid")
    fmt.Fprintf(w, "you are modify user %s", uid)
}

func deleteuser(w http.ResponseWriter, r *http.Request) {
    params := r.URL.Query()
    uid := params.Get(":uid")
    fmt.Fprintf(w, "you are delete user %s", uid)
}

func adduser(w http.ResponseWriter, r *http.Request) {
    uid := r.FormValue("uid")
    fmt.Fprint(w, "you are add user %s", uid)
}

func main() {
    mux := routes.New()
    mux.Get("/user/:uid", getuser)
    mux.Post("/user/", adduser)
    mux.Del("/user/:uid", deleteuser)
    mux.Put("/user/:uid", modifyuser)
    http.Handle("/", mux)
    http.ListenAndServe(":8088", nil)
}
```

上面的代码演示了如何编写一个REST的应用，我们访问的资源是用户，我们通过不同的method来访问不同的函数，这里使用了第三方库 `github.com/drone/routes` ，在前面章节我们介绍过如何实现自定义的路由器，这个库实现了自定义路由和方便的路由规则映射，通过它，我们可以很方便的实现REST的架构。通过上面的代码可知，REST就是根据不同的method访问同一个资源的时候实现不同的逻辑处理。

总结

REST是一种架构风格，汲取了WWW的成功经验：无状态，以资源为中心，充分利用HTTP协议和URI协议，提供统一的接口定义，使得它作为一种设计Web服务的方法而变得流行。在某种意义上，通过强调URI和HTTP等早期Internet标准，REST是对大型应用程序服务器时代之前的Web方式的回归。目前Go对于REST的支持还是很简单的，通过实现自定义的路由规则，我们就可以为不同的method实现不同的handle，这样就实现了REST的架构。

links

- 目录
- 上一节: WebSocket
- 下一节: RPC

8.4 RPC

前面几个小节我们介绍了如何基于Socket和HTTP来编写网络应用，通过学习我们了解了Socket和HTTP采用的是类似"信息交换"模式，即客户端发送一条信息到服务端，然后(一般来说)服务器端都会返回一定的信息以表示响应。客户端和服务端之间约定了交互信息的格式，以便双方都能够解析交互所产生的信息。但是很多独立的应用并没有采用这种模式，而是采用类似常规的函数调用的方式来完成想要的功能。

RPC就是想实现函数调用模式的网络化。客户端就像调用本地函数一样，然后客户端把这些参数打包之后通过网络传递到服务端，服务端解包到处理过程中执行，然后执行的结果反馈给客户端。

RPC（Remote Procedure Call Protocol）——远程过程调用协议，是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。它假定某些传输协议的存在，如TCP或UDP，以便为通信程序之间携带信息数据。通过它可以使函数调用模式网络化。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC工作原理

图8.8 RPC工作流程图

运行时,一次客户机对服务器的RPC调用,其内部操作大致有如下十步：

- 1.调用客户端句柄；执行传送参数
- 2.调用本地系统内核发送网络消息
- 3.消息传送到远程主机
- 4.服务器句柄得到消息并取得参数
- 5.执行远程过程
- 6.执行的过程将结果返回服务器句柄
- 7.服务器句柄返回结果，调用远程系统内核
- 8.消息传回本地主机
- 9.客户句柄由内核接收消息
- 10.客户接收句柄返回的数据

Go RPC

Go标准包中已经提供了对RPC的支持，而且支持三个级别的RPC：TCP、HTTP、JSONRPC。但Go的RPC包是独一无二的RPC，它和传统的RPC系统不同，它只支持Go开发的服务器与客户端之间的交互，因为在内部，它们采用了Gob来编码。

Go RPC的函数只有符合下面的条件才能被远程访问，不然会被忽略，详细的要求如下：

- 函数必须是导出的(首字母大写)
- 必须有两个导出类型的参数，
- 第一个参数是接收的参数，第二个参数是返回给客户端的参数，第二个参数必须是指针类型的
- 函数还要有一个返回值error

举个例子，正确的RPC函数格式如下：

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

T、T1和T2类型必须能被 `encoding/gob` 包编解码。

任何的RPC都需要通过网络来传递数据，Go RPC可以利用HTTP和TCP来传递数据，利用HTTP的好处是可以直接复用 `net/http` 里面的一些函数。详细的例子请看下面的实现

HTTP RPC

http的服务端代码实现如下：

```
package main

import (
    "errors"
    "fmt"
    "net/http"
    "net/rpc"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {

    arith := new(Arith)
    rpc.Register(arith)
    rpc.HandleHTTP()

    err := http.ListenAndServe(":1234", nil)
    if err != nil {
        fmt.Println(err.Error())
    }
}
```

通过上面的例子可以看到，我们注册了一个Arith的RPC服务，然后通过 `rpc.HandleHTTP` 函数把该服务注册到了HTTP协议上，然后我们就可以利用http的方式来传递数据了。

请看下面的客户端代码：

```
package main

import (
    "fmt"
    "log"
    "net/rpc"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: ", os.Args[0], "server")
        os.Exit(1)
    }
    serverAddress := os.Args[1]

    client, err := rpc.DialHTTP("tcp", serverAddress+":1234")
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, qu

}
```

我们把上面的服务端和客户端的代码分别编译，然后先把服务端开启，然后开启客户端，输入代码，就会输出如下信息：

```
$ ./http_c localhost
Arith: 17*8=136
Arith: 17/8=2 remainder 1
```

通过上面的调用可以看到参数和返回值是我们定义的struct类型，在服务端我们把它们当做调用函数的参数的类型，在客户端作为 `client.Call` 的第2，3两个参数的类型。客户端最重要的就是这个Call函数，它有3个参数，第1个要调用的函数的名字，第2个是要传递的参数，第3个要返回的参数(注意是指针类型)，通过上面的代码例子我们可以发现，使用Go的RPC实现相当的简单，方便。

TCP RPC

上面我们实现了基于HTTP协议的RPC，接下来我们要实现基于TCP协议的RPC，服务端的实现代码如下所示：

```
package main

import (
    "errors"
    "fmt"
    "net"
    "net/rpc"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

```
func main() {

    arith := new(Arith)
    rpc.Register(arith)

    tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
    checkError(err)

    listener, err := net.ListenTCP("tcp", tcpAddr)
    checkError(err)

    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        rpc.ServeConn(conn)
    }

}

func checkError(err error) {
    if err != nil {
        fmt.Println("Fatal error ", err.Error())
        os.Exit(1)
    }
}
```

上面这个代码和http的服务器相比，不同在于:在此处我们采用了TCP协议，然后需要自己控制连接，当有客户端连接上来后，我们需要把这个连接交给rpc来处理。

如果你留心了，你会发现这它是一个阻塞型的单用户的程序，如果想要实现多并发，那么可以使用goroutine来实现，我们前面在socket小节的时候已经介绍过如何处理goroutine。 下面展现了TCP实现的RPC客户端：

```
package main

import (
    "fmt"
    "log"
    "net/rpc"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: ", os.Args[0], "server:port")
        os.Exit(1)
    }
    service := os.Args[1]

    client, err := rpc.Dial("tcp", service)
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, qu

}
```

这个客户端代码和http的客户端代码对比，唯一的区别一个是DialHTTP，一个是Dial(tcp)，其他处理一模一样。

JSON RPC

JSON RPC是数据编码采用了JSON，而不是gob编码，其他和上面介绍的RPC概念一模一样，下面我们来演示一下，如何使用Go提供的json-rpc标准包，请看服务端代码的实现：

```
package main

import (
    "errors"
    "fmt"
    "net"
    "net/rpc"
    "net/rpc/jsonrpc"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {

    arith := new(Arith)
    rpc.Register(arith)

    tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
    checkError(err)

    listener, err := net.ListenTCP("tcp", tcpAddr)
    checkError(err)
```

```
    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        jsonrpc.ServeConn(conn)
    }

}

func checkError(err error) {
    if err != nil {
        fmt.Println("Fatal error ", err.Error())
        os.Exit(1)
    }
}
```

通过示例我们可以看出 json-rpc是基于TCP协议实现的，目前它还不支持HTTP方式。

请看客户端的实现代码：

```
package main

import (
    "fmt"
    "log"
    "net/rpc/jsonrpc"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: ", os.Args[0], "server:port")
        log.Fatal(1)
    }
    service := os.Args[1]

    client, err := jsonrpc.Dial("tcp", service)
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, qu

}
```

总结

Go已经提供了对RPC的良好支持，通过上面HTTP、TCP、JSON RPC的实现,我们就可以很方便的开发很多分布式的Web应用，我想作为读者的你已经领会到这一点。但遗憾的是目前Go尚未提供对SOAP RPC的支持，欣慰的是现在已经有第三方的开源实现了。

links

- 目录
- 上一节: REST
- 下一节: 小结

8.5 小结

这一章我们介绍了目前流行的几种主要的网络应用开发方式，第一小节介绍了网络编程中的基础:Socket编程，因为现在网络正在朝云的方向快速进化，作为这一技术演进的基石的的socket知识，作为开发者的你，是必须要掌握的。第二小节介绍了正愈发流行的HTML5中一个重要的特性WebSocket，通过它,服务器可以实现主动的push消息，以简化以前ajax轮询的模式。第三小节介绍了REST编写模式，这种模式特别适合来开发网络应用API，目前移动应用的快速发展，我觉得将来会是一个潮流。第四小节介绍了Go实现的RPC相关知识，对于上面四种开发方式，Go都已经提供了良好的支持，net包及其子包,是所有涉及到网络编程的工具的所在地。如果你想更加深入的了解相关实现细节，可以尝试阅读这个包下面的源码。

links

- 目录
- 上一节: RPC
- 下一章: 安全与加密

9 安全与加密

无论是开发Web应用的开发者还是企图利用Web应用漏洞的攻击者，对于Web程序安全这个话题都给予了越来越多的关注。特别是最近CSDN密码泄露事件，更是让我们对Web安全这个话题更加重视，所有人都谈密码色变，都开始检测自己的系统是否存在漏洞。那么我们作为一名Go程序的开发者，一定也需要知道我们的应用程序随时会成为众多攻击者的目标，并提前做好防范的准备。

很多Web应用程序中的安全问题都是由于轻信了第三方提供的数据造成的。比如对于用户的输入数据，在对其进行验证之前都应该将其视为不安全的数据。如果直接把这些不安全的数据输出到客户端，就可能造成跨站脚本攻击(XSS)的问题。如果把不安全的数据用于数据库查询，那么就可能造成SQL注入问题，我们将会在9.3、9.4小节介绍如何避免这些问题。

在使用第三方提供的数据，包括用户提供的数据时，首先检验这些数据的合法性非常重要，这个过程叫做过滤，我们将在9.2小节介绍如何保证对所有输入的数据进行过滤处理。

过滤输入和转义输出并不能解决所有的安全问题，我们将会在9.1讲解的CSRF攻击，会导致受骗者发送攻击者指定的请求从而造成一些破坏。

与安全加密相关的，能够增强我们的Web应用程序的强大手段就是加密，CSDN泄密事件就是因为密码保存的是明文，使得攻击拿手库之后就可以直接实施一些破坏行为了。不过，和其他工具一样，加密手段也必须运用得当。我们将在9.5小节介绍如何存储密码，如何让密码存储的安全。

加密的本质就是扰乱数据，某些不可恢复的数据扰乱我们称为单向加密或者散列算法。另外还有一种双向加密方式，也就是可以对加密后的数据进行解密。我们将会在9.6小节介绍如何实现这种双向加密方式。

目录

links

- 目录
- 上一章: 第八章总结
- 下一节: 预防CSRF攻击

9.1 预防CSRF攻击

什么是CSRF

CSRF（Cross-site request forgery），中文名称：跨站请求伪造，也被称为：one click attack/session riding，缩写为：CSRF/XSRF。

那么CSRF到底能够干嘛呢？你可以这样简单的理解：攻击者可以盗用你的登陆信息，以你的身份模拟发送各种请求。攻击者只要借助少许的社会工程学的诡计，例如通过QQ等聊天软件发送的链接(有些还伪装成短域名，用户无法分辨)，攻击者就能迫使Web应用的用户去执行攻击者预设的操作。例如，当用户登录网络银行去查看其存款余额，在他没有退出时，就点击了一个QQ好友发来的链接，那么该用户银行帐户中的资金就有可能被转移到攻击者指定的帐户中。

所以遇到CSRF攻击时，将对终端用户的数据和操作指令构成严重的威胁；当受攻击的终端用户具有管理员帐户的时候，CSRF攻击将危及整个Web应用程序。

CSRF的原理

下图简单阐述了CSRF攻击的思想

图9.1 CSRF的攻击过程

从上图可以看出，要完成一次CSRF攻击，受害者必须依次完成两个步骤 ：

- 1.登录受信任网站A，并在本地生成Cookie 。
- 2.在不退出A的情况下，访问危险网站B。

看到这里，读者也许会问：“如果我不满足以上两个条件中的任意一个，就不会受到CSRF的攻击”。是的，确实如此，但你不能保证以下情况不会发生：

- 你不能保证你登录了一个网站后，不再打开一个tab页面并访问另外的网站，特别现在浏览器都是支持多tab的。
- 你不能保证你关闭浏览器了后，你本地的Cookie立刻过期，你上次的会话已经结束。
- 上图中所谓的攻击网站，可能是一个存在其他漏洞的可信任的经常被人访问的网站。

因此对于用户来说很难避免在登陆一个网站之后不点击一些链接进行其他操作，所以随时可能成为CSRF的受害者。

CSRF攻击主要是因为Web的隐式身份验证机制，Web的身份验证机制虽然可以保证一个请求是来自于某个用户的浏览器，但却无法保证该请求是用户批准发送的。

如何预防CSRF

过上面的介绍，读者是否觉得这种攻击很恐怖，意识到恐怖是个好事情，这样会促使你接着往下看如何改进和防止类似的漏洞出现。

CSRF的防御可以从服务端和客户端两方面着手，防御效果是从服务端着手效果比较好，现在一般的CSRF防御也都在服务端进行。

服务端的预防CSRF攻击的方式方法有多种，但思想上都是差不多的，主要从以下2个方面入手：

- 1、正确使用GET,POST和Cookie；
- 2、在非GET请求中增加伪随机数；

我们上一章介绍过REST方式的Web应用，一般而言，普通的Web应用都是以GET、POST为主，还有一种请求是Cookie方式。我们一般都是按照如下方式设计应用：

1、GET常用在查看，列举，展示等不需要改变资源属性的时候；

2、POST常用在下达订单，改变一个资源的属性或者做其他一些事情；

接下来我就以Go语言来举例说明，如何限制对资源的访问方法：

```
mux.Get("/user/:uid", getuser)
mux.Post("/user/:uid", modifyuser)
```

这样处理后，因为我们限定了修改只能使用POST，当GET方式请求时就拒绝响应，所以上面图示中GET方式的CSRF攻击就可以防止了，但这样就能全部解决问题了吗？当然不是，因为POST也是可以模拟的。

因此我们需要实施第二步，在非GET方式的请求中增加随机数，这个大概有三种方式来进行：

- 为每个用户生成一个唯一的cookie token，所有表单都包含同一个伪随机值，这种方案最简单，因为攻击者不能获得第三方的Cookie(理论上)，所以表单中的数据也就构造失败，但是由于用户的Cookie很容易由于网站的XSS漏洞而被盗取，所以这个方案必须要在没有XSS的情况下才安全。
- 每个请求使用验证码，这个方案是完美的，因为要多次输入验证码，所以用户友好性很差，所以不适合实际运用。
- 不同的表单包含一个不同的伪随机值，我们在4.4小节介绍“如何防止表单多次递交”时介绍过此方案，复用相关代码，实现如下：

生成随机数token

```
h := md5.New()
io.WriteString(h, strconv.FormatInt(crutime, 10))
io.WriteString(h, "ganraomaxxxxxxxxx")
token := fmt.Sprintf("%x", h.Sum(nil))

t, _ := template.ParseFiles("login.gtpl")
t.Execute(w, token)
```

输出token

```
<input type="hidden" name="token" value="{{.}}">
```

验证token

```
r.ParseForm()
token := r.Form.Get("token")
if token != "" {
    //验证token的合法性
} else {
    //不存在token报错
}
```

这样基本就实现了安全的POST，但是也许你会说如果破解了token的算法呢，按照理论上是，但是实际上破解是基本不可能的，因为有人曾计算过，暴力破解该串大概需要2的11次方时间。

总结

跨站请求伪造，即CSRF，是一种非常危险的Web安全威胁，它被Web安全界称为“沉睡的巨人”，其威胁程度由此“美誉”便可见一斑。本小节不仅对跨站请求伪造本身进行了简单介绍，还详细说明造成这种漏洞的原因所在，然后以此提了一些防范该攻击的建议，希望对读者编写安全的Web应用能够有所启发。

links

- 目录
- 上一节: 安全与加密
- 下一节: 确保输入过滤

9.2 确保输入过滤

过滤用户数据是Web应用安全的基础。它是验证数据合法性的过程。通过对所有的输入数据进行过滤，可以避免恶意数据在程序中被误信或误用。大多数Web应用的漏洞都是因为没有对用户输入的数据进行恰当过滤所引起的。

我们介绍的过滤数据分成三个步骤：

- 1、识别数据，搞清楚需要过滤的数据来自于哪里
- 2、过滤数据，弄明白我们需要什么样的数据
- 3、区分已过滤及被污染数据，如果存在攻击数据那么保证过滤之后可以让我们使用更安全的数据

识别数据

“识别数据”作为第一步是因为在你不知道“数据是什么，它来自于哪里”的前提下，你也就不能正确地过滤它。这里的数据是指所有源自非代码内部提供的数据。例如:所有来自客户端的数据，但客户端并不是唯一的外部数据源，数据库和第三方提供的接口数据等也可以是外部数据源。

由用户输入的数据我们通过Go非常容易识别，Go通过 `r.ParseForm` 之后，把用户POST和GET的数据全部放在了 `r.Form` 里面。其它的输入要难识别得多，例如， `r.Header` 中的很多元素是由客户端所操纵的。常常很难确认其中的哪些元素组成了输入，所以，最好的方法是把里面所有的数据都看成是用户输入。(例如 `r.Header.Get("Accept-Charset")` 这样的也看做是用户输入,虽然这些大多数是浏览器操纵的)

过滤数据

在知道数据来源之后，就可以过滤它了。过滤是一个有点正式的术语，它在平时表述中有很多同义词，如验证、清洁及净化。尽管这些术语表面意义不同，但它们都是指的同一个处理：防止非法数据进入你的应用。

过滤数据有很多种方法，其中有一些安全性较差。最好的方法是把过滤看成是一个检查的过程，在你使用数据之前都检查一下看它们是否是符合合法数据的要求。而且不要试图好心地去纠正非法数据，而要让用户按你制定的规则去输入数据。历史证明了试图纠正非法数据往往会导致安全漏洞。这里举个例子：“最近建设银行系统升级之后，如果密码后面两位是0，只要输入前面四位就能登录系统”，这是一个非常严重的漏洞。

过滤数据主要采用如下一些库来操作：

- strconv包下面的字符串转化相关函数，因为从Request中的 `r.Form` 返回的是字符串，而有些时候我们需要将之转化成整/浮点数， `Atoi` 、 `ParseBool` 、 `ParseFloat` 、 `ParseInt` 等函数就可以派上

  用场了。
- string包下面的一些过滤函数 `Trim` 、 `ToLower` 、 `ToTitle` 等函数，能够帮助我们按照指定的格式获取信息。
- regexp包用来处理一些复杂的需求，例如判定输入是否是Email、生日之类。

过滤数据除了检查验证之外，在特殊时候，还可以采用白名单。即假定你正在检查的数据都是非法的，除非能证明它是合法的。使用这个方法，如果出现错误，只会导致把合法的数据当成是非法的，而不会是相反，尽管我们不想犯任何错误，但这样总比把非法数据当成合法数据要安全得多。

区分过滤数据

如果完成了上面的两步，数据过滤的工作就基本完成了，但是在编写Web应用的时候我们还需要区分已过滤和被污染数据，因为这样可以保证过滤数据的完整性，而不影响输入的数据。我们约定把所有经过过滤的数据放入一个叫全局的Map变量中(CleanMap)。这时需要用两个重要的步骤来防止被污染数据的注入：

- 每个请求都要初始化CleanMap为一个空Map。
- 加入检查及阻止来自外部数据源的变量命名为CleanMap。

接下来，让我们通过一个例子来巩固这些概念，请看下面这个表单

```
<form action="/whoami" method="POST">
    我是谁:
    <select name="name">
        <option value="astaxie">astaxie</option>
        <option value="herry">herry</option>
        <option value="marry">marry</option>
    </select>
    <input type="submit" />
</form>
```

在处理这个表单的编程逻辑中，非常容易犯的错误是认为只能提交三个选择中的一个。其实攻击者可以模拟POST操作，递交 `name=attack` 这样的数据，所以在此时我们需要做类似白名单的处理

```
r.ParseForm()
name := r.Form.Get("name")
CleanMap := make(map[string]interface{}, 0)
if name == "astaxie" || name == "herry" || name == "marry" {
    CleanMap["name"] = name
}
```

上面代码中我们初始化了一个CleanMap的变量，当判断获取的name是 `astaxie` 、 `herry` 、 `marry` 三个中的一个之后 ，我们把数据存储到了CleanMap之中，这样就可以确保CleanMap["name"]中的数据是合法的，从而在代

码的其它部分使用它。当然我们还可以在else部分增加非法数据的处理，一种可能是再次显示表单并提示错误。但是不要试图为了友好而输出被污染的数据。

上面的方法对于过滤一组已知的合法值的数据很有效，但是对于过滤有一组已知合法字符组成的数据时就没有什么帮助。例如，你可能需要一个用户名只能由字母及数字组成：

```
r.ParseForm()
username := r.Form.Get("username")
CleanMap := make(map[string]interface{}, 0)
if ok, _ := regexp.MatchString("^[a-zA-Z0-9].$", username); ok {
    CleanMap["username"] = username
}
```

总结

数据过滤在Web安全中起到一个基石的作用，大多数的安全问题都是由于没有过滤数据和验证数据引起的，例如前面小节的CSRF攻击，以及接下来将要介绍的XSS攻击、SQL注入等都是没有认真地过滤数据引起的，因此我们需要特别重视这部分的内容。

links

- 目录
- 上一节: 预防CSRF攻击
- 下一节: 避免XSS攻击

9.3 避免XSS攻击

随着互联网技术的发展，现在的Web应用都含有大量的动态内容以提高用户体验。所谓动态内容，就是应用程序能够根据用户环境和用户请求，输出相应的内容。动态站点会受到一种名为“跨站脚本攻击”（Cross Site Scripting, 安全专家们通常将其缩写成 XSS）的威胁，而静态站点则完全不受其影响。

什么是XSS

XSS攻击：跨站脚本攻击(Cross-Site Scripting)，为了不和层叠样式表(Cascading Style Sheets, CSS)的缩写混淆，故将跨站脚本攻击缩写为XSS。XSS是一种常见的web安全漏洞，它允许攻击者将恶意代码植入到提供给其它用户使用的页面中。不同于大多数攻击(一般只涉及攻击者和受害者)，XSS涉及到三方，即攻击者、客户端与Web应用。XSS的攻击目标是为了盗取存储在客户端的cookie或者其他网站用于识别客户端身份的敏感信息。一旦获取到合法用户的信息后，攻击者甚至可以假冒合法用户与网站进行交互。

XSS通常可以分为两大类：一类是存储型XSS，主要出现在让用户输入数据，供其他浏览此页的用户进行查看的地方，包括留言、评论、博客日志和各类表单等。应用程序从数据库中查询数据，在页面中显示出来，攻击者在相关页面输入恶意的脚本数据后，用户浏览此类页面时就可能受到攻击。这个流程简单可以描述为:恶意用户的Html输入Web程序->进入数据库->Web程序->用户浏览器。另一类是反射型XSS，主要做法是将脚本代码加入URL地址的请求参数里，请求参数进入程序后在页面直接输出，用户点击类似的恶意链接就可能受到攻击。

XSS目前主要的手段和目的如下：

- 盗用cookie，获取敏感信息。
- 利用植入Flash，通过crossdomain权限设置进一步获取更高权限；或者利用Java等得到类似的操作。
- 利用iframe、frame、XMLHttpRequest或上述Flash等方式，以（被攻击者）用户的身份执行一些管理动作，或执行一些如:发微博、加好友、发私信等常规操作，前段时间新浪微博就遭遇过一次XSS。
- 利用可被攻击的域受到其他域信任的特点，以受信任来源的身份请求一些平时不允许的操作，如进行不当的投票活动。
- 在访问量极大的一些页面上的XSS可以攻击一些小型网站，实现DDoS攻击的效果

XSS的原理

Web应用未对用户提交请求的数据做充分的检查过滤，允许用户在提交的数据中掺入HTML代码(最主要的是“>”、“<”)，并将未经转义的恶意代码输出到第三方用户的浏览器解释执行，是导致XSS漏洞的产生原因。

接下来以反射性XSS举例说明XSS的过程：现在有一个网站，根据参数输出用户的名称，例如访问url： `http://127.0.0.1/?name=astaxie` ，就会在浏览器输出如下信息：

```
hello astaxie
```

如果我们传递这样的url： `http://127.0.0.1/?name=<script>alert('astaxie,xss&#3` ,这时你就会发现浏览器跳出一个弹出框，这说明站点已经存在了XSS漏洞。那么恶意用户是如何盗取Cookie的呢？与上类似，如下这样的url： `http://127.0.0.1/?name=<script>document.location.href='h` ，这样就可以把当前的cookie发送到指定的站点：www.xxx.com。你也许会说，这样的URL一看就有问题，怎么会有人点击？，是的，这类的URL会让人怀疑，但如果使用短网址服务将之缩短，你还看得出来么？攻击者将缩短过后的url通过某些途径传播开来，不明真相的用户一旦点击了这样的url，相应cookie数据就会被发送事先设定好的站点，这样子就盗得了用户的cookie信息，然后就可以利用Websleuth之类的工具来检查是否能盗取那个用户的账户。

更加详细的关于XSS的分析大家可以参考这篇叫做《新浪微博XSS事件分析》的文章。

如何预防XSS

答案很简单，坚决不要相信用户的任何输入，并过滤掉输入中的所有特殊字符。这样就能消灭绝大部分的XSS攻击。

目前防御XSS主要有如下几种方式：

- 过滤特殊字符

  避免XSS的方法之一主要是将用户所提供的内容进行过滤，Go语言提供了HTML的过滤函数：

  text/template包下面的HTMLEscapeString、JSEscapeString等函数

- 使用HTTP头指定类型

  `w.Header().Set("Content-Type","text/javascript")`

  这样就可以让浏览器解析javascript代码，而不会是html输出。

总结

XSS漏洞是相当有危害的，在开发Web应用的时候，一定要记住过滤数据，特别是在输出到客户端之前，这是现在行之有效的防止XSS的手段。

links

- 目录
- 上一节: 确保输入过滤
- 下一节: 避免SQL注入

9.4 避免SQL注入

什么是SQL注入

SQL注入攻击（SQL Injection），简称注入攻击，是Web开发中最常见的一种安全漏洞。可以用它来从数据库获取敏感信息，或者利用数据库的特性执行添加用户，导出文件等一系列恶意操作，甚至有可能获取数据库乃至系统用户最高权限。

而造成SQL注入的原因是因为程序没有有效过滤用户的输入，使攻击者成功的向服务器提交恶意的SQL查询代码，程序在接收后错误的将攻击者的输入作为查询语句的一部分执行，导致原始的查询逻辑被改变，额外的执行了攻击者精心构造的恶意代码。

SQL注入实例

很多Web开发者没有意识到SQL查询是可以被篡改的，从而把SQL查询当作可信任的命令。殊不知，SQL查询是可以绕开访问控制，从而绕过身份验证和权限检查的。更有甚者，有可能通过SQL查询去运行主机系统级的命令。

下面将通过一些真实的例子来详细讲解SQL注入的方式。

考虑以下简单的登录表单：

```
<form action="/login" method="POST">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" value="登陆" /></p>
</form>
```

我们的处理里面的SQL可能是这样的：

```
username:=r.Form.Get("username")
password:=r.Form.Get("password")
sql:="SELECT * FROM user WHERE username='"+username+"' AND passwor
```

如果用户的输入的用户名如下，密码任意

```
myuser' or 'foo' = 'foo' --
```

那么我们的SQL变成了如下所示：

```
SELECT * FROM user WHERE username='myuser' or 'foo'=='foo' --'' AND
```

在SQL里面 `--` 是注释标记，所以查询语句会在此中断。这就让攻击者在不知道任何合法用户名和密码的情况下成功登录了。

对于MSSQL还有更加危险的一种SQL注入，就是控制系统，下面这个可怕的例子将演示如何在某些版本的MSSQL数据库上执行系统命令。

```
sql:="SELECT * FROM products WHERE name LIKE '%"+prod+"%'"
Db.Exec(sql)
```

如果攻击提交 `a%' exec master..xp_cmdshell 'net user test testpass /ADD' --` 作为变量 prod的值，那么sql将会变成

```
sql:="SELECT * FROM products WHERE name LIKE '%a%' exec master..xp_
```

MSSQL服务器会执行这条SQL语句，包括它后面那个用于向系统添加新用户的命令。如果这个程序是以sa运行而 MSSQLSERVER服务又有足够的权限的话，攻击者就可以获得一个系统帐号来访问主机了。

> 虽然以上的例子是针对某一特定的数据库系统的，但是这并不代表不能对其它数据库系统实施类似的攻击。针对这种安全漏洞，只要使用不同方法，各种数据库都有可能遭殃。

如何预防SQL注入

也许你会说攻击者要知道数据库结构的信息才能实施SQL注入攻击。确实如此，但没人能保证攻击者一定拿不到这些信息，一旦他们拿到了，数据库就存在泄露的危险。如果你在用开放源代码的软件包来访问数据库，比如论坛程序，攻击者就很容易得到相关的代码。如果这些代码设计不良的话，风险就更大了。目前Discuz、phpwind、phpcms等这些流行的开源程序都有被SQL注入攻击的先例。

这些攻击总是发生在安全性不高的代码上。所以，永远不要信任外界输入的数据，特别是来自于用户的数据，包括选择框、表单隐藏域和 cookie。就如上面的第一个例子那样，就算是正常的查询也有可能造成灾难。

SQL注入攻击的危害这么大，那么该如何来防治呢?下面这些建议或许对防治SQL注入有一定的帮助。

1. 严格限制Web应用的数据库的操作权限，给此用户提供仅仅能够满足其工作的最低权限，从而最大限度的减少注入攻击对数据库的危害。
2. 检查输入的数据是否具有所期望的数据格式，严格限制变量的类型，例如使用

regexp包进行一些匹配处理，或者使用strconv包对字符串转化成其他基本类型的数据进行判断。
3. 对进入数据库的特殊字符（'"\尖括号&*;等）进行转义处理，或编码转换。Go 的 `text/template` 包里面的 `HTMLEscapeString` 函数可以对字符串进行转义处理。
4. 所有的查询语句建议使用数据库提供的参数化查询接口，参数化的语句使用参数而不是将用户输入变量嵌入到SQL语句中，即不要直接拼接SQL语句。例如使用 `database/sql` 里面的查询函数 `Prepare` 和 `Query` ，或者 `Exec(query string, args ...interface{})` 。
5. 在应用发布之前建议使用专业的SQL注入检测工具进行检测，以及时修补被发现的SQL注入漏洞。网上有很多这方面的开源工具，例如sqlmap、SQLninja等。
6. 避免网站打印出SQL错误信息，比如类型错误、字段不匹配等，把代码里的SQL语句暴露出来，以防止攻击者利用这些错误信息进行SQL注入。

总结

通过上面的示例我们可以知道，SQL注入是危害相当大的安全漏洞。所以对于我们平常编写的Web应用，应该对于每一个小细节都要非常重视，细节决定命运，生活如此，编写Web应用也是这样。

links

- 目录
- 上一节: 避免XSS攻击
- 下一节: 存储密码

9.5 存储密码

过去一段时间以来, 许多的网站遭遇用户密码数据泄露事件, 这其中包括顶级的互联网企业–Linkedin, 国内诸如CSDN，该事件横扫整个国内互联网，随后又爆出多玩游戏800万用户资料被泄露，另有传言人人网、开心网、天涯社区、世纪佳缘、百合网等社区都有可能成为黑客下一个目标。层出不穷的类似事件给用户的网上生活造成巨大的影响，人人自危，因为人们往往习惯在不同网站使用相同的密码，所以一家“暴库”，全部遭殃。

那么我们作为一个Web应用开发者，在选择密码存储方案时, 容易掉入哪些陷阱, 以及如何避免这些陷阱？

普通方案

目前用的最多的密码存储方案是将明文密码做单向哈希后存储，单向哈希算法有一个特征：无法通过哈希后的摘要(digest)恢复原始数据，这也是“单向”二字的来源。常用的单向哈希算法包括SHA-256, SHA-1, MD5等。

Go语言对这三种加密算法的实现如下所示：

```
//import "crypto/sha256"
h := sha256.New()
io.WriteString(h, "His money is twice tainted: 'taint yours and 'ta
fmt.Printf("% x", h.Sum(nil))

//import "crypto/sha1"
h := sha1.New()
io.WriteString(h, "His money is twice tainted: 'taint yours and 'ta
fmt.Printf("% x", h.Sum(nil))

//import "crypto/md5"
h := md5.New()
io.WriteString(h, "需要加密的密码")
fmt.Printf("%x", h.Sum(nil))
```

单向哈希有两个特性：

- 1）同一个密码进行单向哈希，得到的总是唯一确定的摘要。
- 2）计算速度快。随着技术进步，一秒钟能够完成数十亿次单向哈希计算。

结合上面两个特点，考虑到多数人所使用的密码为常见的组合，攻击者可以将所有密码的常见组合进行单向哈希，得到一个摘要组合, 然后与数据库中的摘要进行比对即可获得对应的密码。这个摘要组合也被称为 `rainbow table` 。

因此通过单向加密之后存储的数据，和明文存储没有多大区别。因此，一旦网站的数据库泄露，所有用户的密码本身就大白于天下。

进阶方案

通过上面介绍我们知道黑客可以用 `rainbow table` 来破解哈希后的密码，很大程度上是因为加密时使用的哈希算法是公开的。如果黑客不知道加密的哈希算法是什么，那他也就无从下手了。

一个直接的解决办法是，自己设计一个哈希算法。然而，一个好的哈希算法是很难设计的——既要避免碰撞，又不能有明显的规律，做到这两点要比想象中的要困难很多。因此实际应用中更多的是利用已有的哈希算法进行多次哈希。

但是单纯的多次哈希，依然阻挡不住黑客。两次 MD5、三次 MD5之类的方法，我们能想到，黑客自然也能想到。特别是对于一些开源代码，这样哈希更是相当于直接把算法告诉了黑客。

没有攻不破的盾，但也没有折不断的矛。现在安全性比较好的网站，都会用一种叫做“加盐”的方式来存储密码，也就是常说的 “salt”。他们通常的做法是，先将用户输入的密码进行一次MD5（或其它哈希算法）加密；将得到的 MD5 值前后加上一些只有管理员自己知道的随机串，再进行一次MD5加密。这个随机串中可以包括某些固定的串，也可以包括用户名（用来保证每个用户加密使用的密钥都不一样）。

```
//import "crypto/md5"
//假设用户名abc，密码123456
h := md5.New()
io.WriteString(h, "需要加密的密码")

//pwmd5等于e10adc3949ba59abbe56e057f20f883e
pwmd5 :=fmt.Sprintf("%x", h.Sum(nil))

//指定两个 salt： salt1 = @#$%   salt2 = ^&*()
salt1 := "@#$%"
salt2 := "^&*()"

//salt1+用户名+salt2+MD5拼接
io.WriteString(h, salt1)
io.WriteString(h, "abc")
io.WriteString(h, salt2)
io.WriteString(h, pwmd5)

last :=fmt.Sprintf("%x", h.Sum(nil))
```

在两个salt没有泄露的情况下，黑客如果拿到的是最后这个加密串，就几乎不可能推算出原始的密码是什么了。

专家方案

上面的进阶方案在几年前也许是足够安全的方案，因为攻击者没有足够的资源建立这么多的 `rainbow table` 。 但是，时至今日，因为并行计算能力的提升，这种攻击已经完全可行。

怎么解决这个问题呢？只要时间与资源允许，没有破译不了的密码，所以方案是:故意增加密码计算所需耗费的资源和时间，使得任何人都不可获得足够的资源建立所需的 `rainbow table` 。

这类方案有一个特点，算法中都有个因子，用于指明计算密码摘要所需要的资源和时间，也就是计算强度。计算强度越大，攻击者建立 `rainbow table` 越困难，以至于不可继续。

这里推荐 `scrypt` 方案，scrypt是由著名的FreeBSD黑客Colin Percival为他的备份服务Tarsnap开发的。

目前Go语言里面支持的库http://code.google.com/p/go/source/browse?repo=crypto#hg%2Fscrypt

```
dk := scrypt.Key([]byte("some password"), []byte(salt), 16384, 8, 1
```

通过上面的的方法可以获取唯一的相应的密码值，这是目前为止最难破解的。

总结

看到这里，如果你产生了危机感，那么就行动起来：

- 1）如果你是普通用户，那么我们建议使用LastPass进行密码存储和生成，对不同的网站使用不同的密码；
- 2）如果你是开发人员， 那么我们强烈建议你采用专家方案进行密码存储。

links

- 目录
- 上一节: 确保输入过滤
- 下一节: 加密和解密数据

9.6 加密和解密数据

前面小节介绍了如何存储密码，但是有的时候，我们想把一些敏感数据加密后存储起来，在将来的某个时候，随需将它们解密出来，此时我们应该在选用对称加密算法来满足我们的需求。

base64加解密

如果Web应用足够简单，数据的安全性没有那么严格的要求，那么可以采用一种比较简单的加解密方法是 `base64` ，这种方式实现起来比较简单，Go语言的 `base64` 包已经很好的支持了这个，请看下面的例子：

```
package main

import (
    "encoding/base64"
    "fmt"
)

func base64Encode(src []byte) []byte {
    return []byte(base64.StdEncoding.EncodeToString(src))
}

func base64Decode(src []byte) ([]byte, error) {
    return base64.StdEncoding.DecodeString(string(src))
}

func main() {
    // encode
    hello := "你好，世界！ hello world"
    debyte := base64Encode([]byte(hello))
    fmt.Println(debyte)
    // decode
    enbyte, err := base64Decode(debyte)
    if err != nil {
        fmt.Println(err.Error())
    }

    if hello != string(enbyte) {
        fmt.Println("hello is not equal to enbyte")
    }

    fmt.Println(string(enbyte))
}
```

高级加解密

Go语言的`crypto`里面支持对称加密的高级加解密包有：

- `crypto/aes`包：AES(Advanced Encryption Standard)，又称Rijndael加密法，是美国联邦政府采用的一种区块加密标准。
- `crypto/des`包：DES(Data Encryption Standard)，是一种对称加密标准，是目前使用最广泛的密钥系统，特别是在保护金融数据的安全中。曾是美国联邦政府的加密标准，但现已被AES所替代。

因为这两种算法使用方法类似，所以在此，我们仅用aes包为例来讲解它们的使用，请看下面的例子

```
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "fmt"
    "os"
)

var commonIV = []byte{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x0

func main() {
    //需要去加密的字符串
    plaintext := []byte("My name is Astaxie")
    //如果传入加密串的话，plaint就是传入的字符串
    if len(os.Args) > 1 {
        plaintext = []byte(os.Args[1])
    }

    //aes的加密字符串
    key_text := "astaxie12798akljzmknm.ahkjkljl;k"
    if len(os.Args) > 2 {
        key_text = os.Args[2]
    }

    fmt.Println(len(key_text))

    // 创建加密算法aes
    c, err := aes.NewCipher([]byte(key_text))
    if err != nil {
        fmt.Printf("Error: NewCipher(%d bytes) = %s", len(key_text
        os.Exit(-1)
    }

    //加密字符串
    cfb := cipher.NewCFBEncrypter(c, commonIV)
    ciphertext := make([]byte, len(plaintext))
    cfb.XORKeyStream(ciphertext, plaintext)
    fmt.Printf("%s=>%x\n", plaintext, ciphertext)

    // 解密字符串
    cfbdec := cipher.NewCFBDecrypter(c, commonIV)
    plaintextCopy := make([]byte, len(plaintext))
    cfbdec.XORKeyStream(plaintextCopy, ciphertext)
    fmt.Printf("%x=>%s\n", ciphertext, plaintextCopy)
}
```

上面通过调用函数 `aes.NewCipher` (参数key必须是16、24或者32位的[]byte，分别对应AES-128, AES-192或AES-256算法),返回了一个 `cipher.Block` 接口，这个接口实现了三个功能：

```
type Block interface {
    // BlockSize returns the cipher's block size.
    BlockSize() int

    // Encrypt encrypts the first block in src into dst.
    // Dst and src may point at the same memory.
    Encrypt(dst, src []byte)

    // Decrypt decrypts the first block in src into dst.
    // Dst and src may point at the same memory.
    Decrypt(dst, src []byte)
}
```

这三个函数实现了加解密操作，详细的操作请看上面的例子。

总结

这小节介绍了几种加解密的算法，在开发Web应用的时候可以根据需求采用不同的方式进行加解密，一般的应用可以采用base64算法，更加高级的话可以采用aes或者des算法。

links

- 目录
- 上一节: 存储密码
- 下一节: 小结

9.7 小结

这一章主要介绍了如：CSRF攻击、XSS攻击、SQL注入攻击等一些Web应用中典型的攻击手法，它们都是由于应用对用户的输入没有很好的过滤引起的，所以除了介绍攻击的方法外，我们也介绍了了如何有效的进行数据过滤，以防止这些攻击的发生的方法。然后针对日异严重的密码泄漏事件，介绍了在设计Web应用中可采用的从基本到专家的加密方案。最后针对敏感数据的加解密简要介绍了，Go语言提供三种对称加密算法：base64、aes和des的实现。

编写这一章的目的是希望读者能够在意识里面加强安全概念，在编写Web应用的时候多留心一点，以使我们编写的Web应用能远离黑客们的攻击。Go语言在支持防攻击方面已经提供大量的工具包，我们可以充分的利用这些包来做出一个安全的Web应用。

links

- 目录
- 上一节: 加密和解密数据
- 下一节: 国际化和本地化

10 国际化和本地化

为了适应经济的全球一体化，作为开发者，我们需要开发出支持多国语言、国际化的Web应用，即同样的页面在不同的语言环境下需要显示不同的效果，也就是说应用程序在运行时能够根据请求所来自的地域与语言的不同而显示不同的用户界面。这样，当需要在应用程序中添加对新的语言的支持时，无需修改应用程序的代码，只需要增加语言包即可实现。

国际化与本地化（Internationalization and localization,通常用i18n和L10N表示），国际化是将针对某个地区设计的程序进行重构，以使它能够在更多地区使用，本地化是指在一个面向国际化的程序中增加对新地区的支持。

目前，Go语言的标准包没有提供对i18n的支持，但有一些比较简单的第三方实现，这一章我们将实现一个go-i18n库，用来支持Go语言的i18n。

所谓的国际化：就是根据特定的locale信息，提取与之相应的字符串或其它一些东西（比如时间和货币的格式）等等。这涉及到三个问题：

1、如何确定locale。

2、如何保存与locale相关的字符串或其它信息。

3、如何根据locale提取字符串和其它相应的信息。

在第一小节里，我们将介绍如何设置正确的locale以便让访问站点的用户能够获得与其语言相应的页面。第二小节将介绍如何处理或存储字符串、货币、时间日期等与locale相关的信息，第三小节将介绍如何实现国际化站点，即如何根据不同locale返回不同合适的内容。通过这三个小节的学习，我们将获得一个完整的i18n方案。

目录

links

- 目录
- 上一章: 第九章总结
- 下一节: 设置默认地区

10.1 设置默认地区

什么是Locale

Locale是一组描述世界上某一特定区域文本格式和语言习惯的设置的集合。locale名通常由三个部分组成：第一部分，是一个强制性的，表示语言的缩写，例如"en"表示英文或"zh"表示中文。第二部分，跟在一个下划线之后，是一个可选的国家说明符，用于区分讲同一种语言的不同国家，例如"en_US"表示美国英语，而"en_UK"表示英国英语。最后一部分，跟在一个句点之后，是可选的字符集说明符，例如"zh_CN.gb2312"表示中国使用gb2312字符集。

GO语言默认采用"UTF-8"编码集，所以我们实现i18n时不考虑第三部分，接下来我们都采用locale描述的前面两部分来作为i18n标准的locale名。

> 在Linux和Solaris系统中可以通过`locale -a`命令列举所有支持的地区名，读者可以看到这些地区名的命名规范。对于BSD等系统，没有locale命令，但是地区信息存储在/usr/share/locale中。

设置Locale

有了上面对locale的定义，那么我们就需要根据用户的信息(访问信息、个人信息、访问域名等)来设置与之相关的locale，我们可以通过如下几种方式来设置用户的locale。

通过域名设置Locale

设置Locale的办法这一就是在应用运行的时候采用域名分级的方式，例如，我们采用www.asta.com当做我们的英文站(默认站)，而把域名www.asta.cn当做中文站。这样通过在应用里面设置域名和相应的locale的对应关系，就可以设置好地区。这样处理有几点好处：

- 通过URL就可以很明显的识别
- 用户可以通过域名很直观的知道将访问那种语言的站点
- 在Go程序中实现非常的简单方便，通过一个map就可以实现
- 有利于搜索引擎抓取，能够提高站点的SEO

我们可以通过下面的代码来实现域名的对应locale：

```
if r.Host == "www.asta.com" {
    i18n.SetLocale("en")
} else if r.Host == "www.asta.cn" {
    i18n.SetLocale("zh-CN")
} else if r.Host == "www.asta.tw" {
    i18n.SetLocale("zh-TW")
}
```

当然除了整域名设置地区之外，我们还可以通过子域名来设置地区，例如"en.asta.com"表示英文站点，"cn.asta.com"表示中文站点。实现代码如下所示：

```
prefix := strings.Split(r.Host,".")

if prefix[0] == "en" {
    i18n.SetLocale("en")
} else if prefix[0] == "cn" {
    i18n.SetLocale("zh-CN")
} else if prefix[0] == "tw" {
    i18n.SetLocale("zh-TW")
}
```

通过域名设置Locale有如上所示的优点，但是我们一般开发Web应用的时候不会采用这种方式，因为首先域名成本比较高，开发一个Locale就需要一个域名，而且往往统一名称的域名不一定能申请的到，其次我们不愿意为每个站点去本地化一个配置，而更多的是采用url后面带参数的方式，请看下面的介绍。

从域名参数设置Locale

目前最常用的设置Locale的方式是在URL里面带上参数，例如www.asta.com/hello?locale=zh或者www.asta.com/zh/hello。这样我们就可以设置地区：`i18n.SetLocale(params["locale"])`。

这种设置方式几乎拥有前面讲的通过域名设置Locale的所有优点，它采用RESTful的方式，以使得我们不需要增加额外的方法来处理。但是这种方式需要在每一个的link里面增加相应的参数locale，这也许有点复杂而且有时候甚至相当的繁琐。不过我们可以写一个通用的函数url，让所有的link地址都通过这个函数来生成，然后在这个函数里面增加`locale=params["locale"]`参数来缓解一下。

也许我们希望URL地址看上去更加的RESTful一点，例如：www.asta.com/en/books(英文站点)和www.asta.com/zh/books(中文站点)，这种方式的URL更加有利于SEO，而且对于用户也比较友好，能够通过URL直观的知道访问的站点。那么这样的URL地址可以通过router来获取locale(参考REST小节里面介绍的router插件实现)：

```
mux.Get("/:locale/books", listbook)
```

从客户端设置地区

在一些特殊的情况下，我们需要根据客户端的信息而不是通过URL来设置Locale，这些信息可能来自于客户端设置的喜好语言(浏览器中设置)，用户的IP地址，用户在注册的时候填写的所在地信息等。这种方式比较适合Web为基础的应用。

- Accept-Language

客户端请求的时候在HTTP头信息里面有 `Accept-Language` ，一般的客户端都会设置该信息，下面是Go语言实现的一个简单的根据 `Accept-Language` 实现设置地区的代码：

```
AL := r.Header.Get("Accept-Language")
if AL == "en" {
    i18n.SetLocale("en")
} else if AL == "zh-CN" {
    i18n.SetLocale("zh-CN")
} else if AL == "zh-TW" {
    i18n.SetLocale("zh-TW")
}
```

当然在实际应用中，可能需要更加严格的判断来进行设置地区

- IP地址

  另一种根据客户端来设定地区就是用户访问的IP，我们根据相应的IP库，对应访问的IP到地区，目前全球比较常用的就是GeoIP Lite Country这个库。这种设置地区的机制非常简单，我们只需要根据IP数据库查询用户的IP然后返回国家地区，根据返回的结果设置对应的地区。

- 用户profile

  当然你也可以让用户根据你提供的下拉菜单或者别的什么方式的设置相应的locale，然后我们将用户输入的信息，保存到与它帐号相关的profile中，当用户再次登陆的时候把这个设置复写到locale设置中，这样就可以保证该用户每次访问都是基于自己先前设置的locale来获得页面。

总结

通过上面的介绍可知，设置Locale可以有很多种方式，我们应该根据需求的不同来选择不同的设置Locale的方法，以让用户能以它最熟悉的方式，获得我们提供的服务，提高应用的用户友好性。

links

- 目录
- 上一节: 国际化和本地化

- 下一节: 本地化资源

10.2 本地化资源

前面小节我们介绍了如何设置Locale，设置好Locale之后我们需要解决的问题就是如何存储相应的Locale对应的信息呢？这里面的信息包括：文本信息、时间和日期、货币值、图片、包含文件以及视图等资源。那么接下来我们将对这些信息一一进行介绍，Go语言中我们把这些格式信息存储在JSON中，然后通过合适的方式展现出来。(接下来以中文和英文两种语言对比举例,存储格式文件en.json和zh-CN.json)

本地化文本消息

本信息是编写Web应用中最常用到的，也是本地化资源中最多的信息，想要以适合本地语言的方式来显示文本信息，可行的一种方案是:建立需要的语言相应的map来维护一个key-value的关系，在输出之前按需从适合的map中去获取相应的文本，如下是一个简单的示例：

```
package main

import "fmt"

var locales map[string]map[string]string

func main() {
    locales = make(map[string]map[string]string, 2)
    en := make(map[string]string, 10)
    en["pea"] = "pea"
    en["bean"] = "bean"
    locales["en"] = en
    cn := make(map[string]string, 10)
    cn["pea"] = "豌豆"
    cn["bean"] = "毛豆"
    locales["zh-CN"] = cn
    lang := "zh-CN"
    fmt.Println(msg(lang, "pea"))
    fmt.Println(msg(lang, "bean"))
}

func msg(locale, key string) string {
    if v, ok := locales[locale]; ok {
        if v2, ok := v[key]; ok {
            return v2
        }
    }
    return ""
}
```

上面示例演示了不同locale的文本翻译，实现了中文和英文对于同一个key显示不同语言的实现，上面实现了中文的文本消息，如果想切换到英文版本，只需要把lang设置为en即可。

有些时候仅是key-value替换是不能满足需要的，例如"I am 30 years old",中文表达是"我今年30岁了"，而此处的30是一个变量，该怎么办呢？这个时候，我们可以结合 `fmt.Printf` 函数来实现，请看下面的代码：

```
en["how old"] ="I am %d years old"
cn["how old"] ="我今年%d岁了"

fmt.Printf(msg(lang, "how old"), 30)
```

上面的示例代码仅用以演示内部的实现方案，而实际数据是存储在JSON里面的，所以我们可以通过 `json.Unmarshal` 来为相应的map填充数据。

本地化日期和时间

因为时区的关系，同一时刻，在不同的地区，表示是不一样的，而且因为Locale的关系，时间格式也不尽相同，例如中文环境下可能显示： `2012年10月24日 星期三 23时11分13秒 CST` ，而在英文环境下可能显示: `Wed Oct 24 23:11:13 CST 2012` 。这里面我们需要解决两点:

1. 时区问题
2. 格式问题

$GOROOT/lib/time包中的timeinfo.zip含有locale对应的时区的定义，为了获得对应于当前locale的时间，我们应首先使用 `time.LoadLocation(name string)` 获取相应于地区的locale，比如 `Asia/Shanghai` 或 `America/Chicago` 对应的时区信息，然后再利用此信息与调用 `time.Now` 获得的Time对象协作来获得最终的时间。详细的请看下面的例子(该例子采用上面例子的一些变量):

```
en["time_zone"]="America/Chicago"
cn["time_zone"]="Asia/Shanghai"

loc,_:=time.LoadLocation(msg(lang,"time_zone"))
t:=time.Now()
t = t.In(loc)
fmt.Println(t.Format(time.RFC3339))
```

我们可以通过类似处理文本格式的方式来解决时间格式的问题，举例如下:

```
en["date_format"]="%Y-%m-%d %H:%M:%S"
cn["date_format"]="%Y年%m月%d日 %H时%M分%S秒"

fmt.Println(date(msg(lang,"date_format"),t))

func date(fomate string,t time.Time) string{
    year, month, day = t.Date()
    hour, min, sec = t.Clock()
    //解析相应的%Y %m %d %H %M %S然后返回信息
    //%Y 替换成2012
    //%m 替换成10
    //%d 替换成24
}
```

本地化货币值

各个地区的货币表示也不一样，处理方式也与日期差不多，细节请看下面代码:

```
en["money"] ="USD %d"
cn["money"] ="￥%d元"

fmt.Println(date(msg(lang,"date_format"),100))

func money_format(fomate string,money int64) string{
    return fmt.Sprintf(fomate,money)
}
```

本地化视图和资源

我们可能会根据Locale的不同来展示视图，这些视图包含不同的图片、css、js等各种静态资源。那么应如何来处理这些信息呢？首先我们应按locale来组织文件信息，请看下面的文件目录安排：

```
views
|--en  //英文模板
    |--images     //存储图片信息
    |--js         //存储JS文件
    |--css        //存储css文件
    index.tpl     //用户首页
    login.tpl     //登陆首页
|--zh-CN //中文模板
    |--images
    |--js
    |--css
    index.tpl
    login.tpl
```

有了这个目录结构后我们就可以在渲染的地方这样来实现代码：

```
s1, _ := template.ParseFiles("views"+lang+"index.tpl")
VV.Lang=lang
s1.Execute(os.Stdout, VV)
```

而对于里面的index.tpl里面的资源设置如下：

```
// js文件
<script type="text/javascript" src="views/{{.VV.Lang}}/js/jquery/jq
// css文件
<link href="views/{{.VV.Lang}}/css/bootstrap-responsive.min.css" re
// 图片文件
<img src="views/{{.VV.Lang}}/images/btn.png">
```

采用这种方式来本地化视图以及资源时，我们就可以很容易的进行扩展了。

总结

本小节介绍了如何使用及存储本地资源，有时需要通过转换函数来实现，有时通过lang来设置，但是最终都是通过key-value的方式来存储Locale对应的数据，在需要时取出相应于Locale的信息后，如果是文本信息就直接输出，如果是时间日期或者货币，则需要先通过`fmt.Printf`或其他格式化函数来处理，而对于不同Locale的视图和资源则是最简单的，只要在路径里面增加lang就可以实现了。

links

- 目录
- 上一节: 设置默认地区

- 下一节: 国际化站点

10.3 国际化站点

前面小节介绍了如何处理本地化资源，即Locale一个相应的配置文件，那么如果处理多个的本地化资源呢？而对于一些我们经常用到的例如：简单的文本翻译、时间日期、数字等如果处理呢？本小节将一一解决这些问题。

管理多个本地包

在开发一个应用的时候，首先我们要决定是只支持一种语言，还是多种语言，如果要支持多种语言，我们则需要制定一个组织结构，以方便将来更多语言的添加。在此我们设计如下：Locale有关的文件放置在config/locales下，假设你要支持中文和英文，那么你需要在这个文件夹下放置en.json和zh.json。大概的内容如下所示：

```
# zh.json

{
"zh": {
    "submit": "提交",
    "create": "创建"
    }
}

#en.json

{
"en": {
    "submit": "Submit",
    "create": "Create"
    }
}
```

为了支持国际化，在此我们使用了一个国际化相关的包——go-i18n，首先我们向go-i18n包注册config/locales这个目录,以加载所有的locale文件

```
Tr:=i18n.NewLocale()
Tr.LoadPath("config/locales")
```

这个包使用起来很简单，你可以通过下面的方式进行测试：

```
fmt.Println(Tr.Translate("submit"))
//输出Submit
Tr.SetLocale("zn")
fmt.Println(Tr.Translate("submit"))
//输出“递交”
```

自动加载本地包

上面我们介绍了如何自动加载自定义语言包，其实go-i18n库已经预加载了很多默认的格式信息，例如时间格式、货币格式，用户可以在自定义配置时改写这些默认配置，请看下面的处理过程：

```
//加载默认配置文件，这些文件都放在go-i18n/locales下面

//文件命名zh.json、en-json、en-US.json等，可以不断的扩展支持更多的语言

func (il *IL) loadDefaultTranslations(dirPath string) error {
    dir, err := os.Open(dirPath)
    if err != nil {
        return err
    }
    defer dir.Close()

    names, err := dir.Readdirnames(-1)
    if err != nil {
        return err
    }

    for _, name := range names {
        fullPath := path.Join(dirPath, name)

        fi, err := os.Stat(fullPath)
        if err != nil {
            return err
        }

        if fi.IsDir() {
            if err := il.loadTranslations(fullPath); err != nil {
                return err
            }
        } else if locale := il.matchingLocaleFromFileName(name); lo
            file, err := os.Open(fullPath)
            if err != nil {
                return err
            }
            defer file.Close()

            if err := il.loadTranslation(file, locale); err != nil
                return err
            }
        }
    }

    return nil
}
```

通过上面的方法加载配置信息到默认的文件，这样我们就可以在我们没有自定义时间信息的时候执行如下的代码获取对应的信息:

```
//locale=zh的情况下，执行如下代码：

fmt.Println(Tr.Time(time.Now()))
//输出：2009年1月08日 星期四 20:37:58 CST

fmt.Println(Tr.Time(time.Now(),"long"))
//输出：2009年1月08日

fmt.Println(Tr.Money(11.11))
//输出:￥11.11
```

template mapfunc

上面我们实现了多个语言包的管理和加载，而一些函数的实现是基于逻辑层的，例如："Tr.Translate"、"Tr.Time"、"Tr.Money"等，虽然我们在逻辑层可以利用这些函数把需要的参数进行转换后在模板层渲染的时候直接输出，但是如果我们想在模版层直接使用这些函数该怎么实现呢？不知你是否还记得，在前面介绍模板的时候说过：Go语言的模板支持自定义模板函数，下面是我们实现的方便操作的mapfunc：

1. 文本信息

文本信息调用 `Tr.Translate` 来实现相应的信息转换，mapFunc的实现如下：

```
func I18nT(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    return Tr.Translate(s)
}
```

注册函数如下：

```
t.Funcs(template.FuncMap{"T": I18nT})
```

模板中使用如下：

```
{{.V.Submit | T}}
```

1. 时间日期

时间日期调用 `Tr.Time` 函数来实现相应的时间转换，mapFunc的实现如下：

```
func I18nTimeDate(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    return Tr.Time(s)
}
```

注册函数如下：

```
t.Funcs(template.FuncMap{"TD": I18nTimeDate})
```

模板中使用如下：

```
{{.V.Now | TD}}
```

1. 货币信息

货币调用 `Tr.Money` 函数来实现相应的时间转换，mapFunc的实现如下：

```
func I18nMoney(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    return Tr.Money(s)
}
```

注册函数如下：

```
t.Funcs(template.FuncMap{"M": I18nMoney})
```

模板中使用如下：

```
{{.V.Money | M}}
```

总结

通过这小节我们知道了如何实现一个多语言包的Web应用，通过自定义语言包我们可以方便的实现多语言，而且通过配置文件能够非常方便的扩充多语言，默认情况下，go-i18n会自定加载一些公共的配置信息，例如时间、货币等，我们就可以非常方便的使用，同时为了支持在模板中使用这些函数，也实现了相应的模板函数，这样就允许我们在开发Web应用的时候直接在模板中通过pipeline的方式来操作多语言包。

links

- 目录
- 上一节: 本地化资源
- 下一节: 小结

10.4 小结

通过这一章的介绍，读者应该对如何操作i18n有了深入的了解，我也根据这一章介绍的内容实现了一个开源的解决方案go-i18n：https://github.com/astaxie/go-i18n 通过这个开源库我们可以很方便的实现多语言版本的Web应用，使得我们的应用能够轻松的实现国际化。如果你发现这个开源库中的错误或者那些缺失的地方，请一起参与到这个开源项目中来，让我们的这个库争取成为Go的标准库。

links

- 目录
- 上一节: 国际化站点
- 下一节: 错误处理，故障排除和测试

11 错误处理，调试和测试

我们经常会看到很多程序员大部分的"编程"时间都花费在检查bug和修复bug上。无论你是在编写修改代码还是重构系统，几乎都是花费大量的时间在进行故障排除和测试，外界都觉得我们程序员是设计师，能够把一个系统从无做到有，是一项很伟大的工作，而且是相当有趣的工作，但事实上我们每天都是徘徊在排错、调试、测试之间。当然如果你有良好的习惯和技术方案来直面这些问题，那么你就有可能将排错时间减到最少，而尽可能的将时间花费在更有价值的事情上。

但是遗憾的是很多程序员不愿意在错误处理、调试和测试能力上下工夫，导致后面应用上线之后查找错误、定位问题花费更多的时间。所以我们在设计应用之前就做好错误处理规划、测试用例等，那么将来修改代码、升级系统都将变得简单。

开发Web应用过程中，错误自然难免，那么如何更好的找到错误原因，解决问题呢？11.1小节将介绍Go语言中如何处理错误，如何设计自己的包、函数的错误处理，11.2小节将介绍如何使用GDB来调试我们的程序，动态运行情况下各种变量信息，运行情况的监控和调试。

11.3小节将对Go语言中的单元测试进行深入的探讨，并示例如何来编写单元测试，Go的单元测试规则规范如何定义，以保证以后升级修改运行相应的测试代码就可以进行最小化的测试。

长期以来，培养良好的调试、测试习惯一直是很多程序员逃避的事情，所以现在你不要再逃避了，就从你现在的项目开发，从学习Go Web开发开始养成良好的习惯。

目录

links

- 目录
- 上一章: 第十章总结
- 下一节: 错误处理

11.1 错误处理

Go语言主要的设计准则是：简洁、明白，简洁是指语法和C类似，相当的简单，明白是指任何语句都是很明显的，不含有任何隐含的东西，在错误处理方案的设计中也贯彻了这一思想。我们知道在C语言里面是通过返回-1或者NULL之类的信息来表示错误，但是对于使用者来说，不查看相应的API说明文档，根本搞不清楚这个返回值究竟代表什么意思，比如:返回0是成功，还是失败,而Go定义了一个叫做error的类型，来显式表达错误。在使用时，通过把返回的error变量与nil的比较，来判定操作是否成功。例如 `os.Open` 函数在打开文件失败时将返回一个不为nil的error变量

```
func Open(name string) (file *File, err error)
```

下面这个例子通过调用 `os.Open` 打开一个文件，如果出现错误，那么就会调用 `log.Fatal` 来输出错误信息：

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
```

类似于 `os.Open` 函数，标准包中所有可能出错的API都会返回一个error变量，以方便错误处理，这个小节将详细地介绍error类型的设计，和讨论开发Web应用中如何更好地处理error。

Error类型

error类型是一个接口类型，这是它的定义：

```
type error interface {
    Error() string
}
```

error是一个内置的接口类型，我们可以在/builtin/包下面找到相应的定义。而我们在很多内部包里面用到的 error是errors包下面的实现的私有结构errorString

```
// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

你可以通过 `errors.New` 把一个字符串转化为errorString，以得到一个满足接口error的对象，其内部实现如下：

```
// New returns an error that formats as the given text.
func New(text string) error {
    return &errorString{text}
}
```

下面这个例子演示了如何使用 `errors.New`：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number'
    }
    // implementation
}
```

在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
f, err := Sqrt(-1)
if err != nil {
    fmt.Println(err)
}
```

自定义Error

通过上面的介绍我们知道error是一个interface，所以在实现自己的包的时候，通过定义实现此接口的结构，我们就可以实现自己的错误定义，请看来自Json包的示例：

```
type SyntaxError struct {
    msg    string // 错误描述
    Offset int64  // 错误发生的位置
}

func (e *SyntaxError) Error() string { return e.msg }
```

Offset字段在调用Error的时候不会被打印，但是我们可以通过类型断言获取错误类型，然后可以打印相应的错误信息，请看下面的例子:

```
if err := dec.Decode(&val); err != nil {
    if serr, ok := err.(*json.SyntaxError); ok {
        line, col := findLine(f, serr.Offset)
        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
    }
    return err
}
```

需要注意的是，函数返回自定义错误时，返回值推荐设置为error类型，而非自定义错误类型，特别需要注意的是不应预声明自定义错误类型的变量。例如：

```
func Decode() *SyntaxError { // 错误，将可能导致上层调用者err!=nil的判断
    var err *SyntaxError     // 预声明错误变量
    if 出错条件 {
        err = &SyntaxError{}
    }
    return err               // 错误，err永远等于非nil，导致上层调用者er
}
```

原因见 http://golang.org/doc/faq#nil_error

上面例子简单的演示了如何自定义Error类型。但是如果我们还需要更复杂的错误处理呢？此时，我们来参考一下net包采用的方法：

```
package net

type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

在调用的地方，通过类型断言err是不是net.Error,来细化错误的处理，例如下面的例子，如果一个网络发生临时性错误，那么将会sleep 1秒之后重试：

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

错误处理

Go在错误处理上采用了与C类似的检查返回值的方式，而不是其他多数主流语言采用的异常方式，这造成了代码编写上的一个很大的缺点:错误处理代码的冗余，对于这种情况是我们通过复用检测函数来减少类似的代码。

请看下面这个例子代码：

```
func init() {
    http.HandleFunc("/view", viewRecord)
}

func viewRecord(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        http.Error(w, err.Error(), 500)
    }
}
```

上面的例子中获取数据和模板展示调用时都有检测错误，当有错误发生时，调用了统一的处理函数 `http.Error` ，返回给客户端500错误码，并显示相应的错误数据。但是当越来越多的HandleFunc加入之后，这样的错误处理逻辑代码就会越来越多，其实我们可以通过自定义路由器来缩减代码(实现的思路可以参考第三章的HTTP详解)。

```
type appHandler func(http.ResponseWriter, *http.Request) error

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Reque
    if err := fn(w, r); err != nil {
        http.Error(w, err.Error(), 500)
    }
}
```

上面我们定义了自定义的路由器，然后我们可以通过如下方式来注册函数：

```
func init() {
    http.Handle("/view", appHandler(viewRecord))
}
```

当请求/view的时候我们的逻辑处理可以变成如下代码，和第一种实现方式相比较已经简单了很多。

```
func viewRecord(w http.ResponseWriter, r *http.Request) error {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return err
    }
    return viewTemplate.Execute(w, record)
}
```

上面的例子错误处理的时候所有的错误返回给用户的都是500错误码，然后打印出来相应的错误代码，其实我们可以把这个错误信息定义的更加友好，调试的时候也方便定位问题，我们可以自定义返回的错误类型：

```
type appError struct {
    Error   error
    Message string
    Code    int
}
```

这样我们的自定义路由器可以改成如下方式：

```
type appHandler func(http.ResponseWriter, *http.Request) *appError

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Reque
    if e := fn(w, r); e != nil { // e is *appError, not os.Error.
        c := appengine.NewContext(r)
        c.Errorf("%v", e.Error)
        http.Error(w, e.Message, e.Code)
    }
}
```

这样修改完自定义错误之后，我们的逻辑处理可以改成如下方式：

```
func viewRecord(w http.ResponseWriter, r *http.Request) *appError
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return &appError{err, "Record not found", 404}
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        return &appError{err, "Can't display record", 500}
    }
    return nil
}
```

如上所示，在我们访问view的时候可以根据不同的情况获取不同的错误码和错误信息，虽然这个和第一个版本的代码量差不多，但是这个显示的错误更加明显，提示的错误信息更加友好，扩展性也比第一个更好。

总结

在程序设计中，容错是相当重要的一部分工作，在Go中它是通过错误处理来实现的，error虽然只是一个接口，但是其变化却可以有很多，我们可以根据自己的需求来实现不同的处理，最后介绍的错误处理方案，希望能给大家在如何设计更好Web错误处理方案上带来一点思路。

links

- 目录
- 上一节: 错误处理，调试和测试
- 下一节: 使用GDB调试

11.2 使用GDB调试

开发程序过程中调试代码是开发者经常要做的一件事情，Go语言不像PHP、Python等动态语言，只要修改不需要编译就可以直接输出，而且可以动态的在运行环境下打印数据。当然Go语言也可以通过Println之类的打印数据来调试，但是每次都需要重新编译，这是一件相当麻烦的事情。我们知道在Python中有pdb/ipdb之类的工具调试，Javascript也有类似工具，这些工具都能够动态的显示变量信息，单步调试等。不过庆幸的是Go也有类似的工具支持：GDB。Go内部已经内置支持了GDB，所以，我们可以通过GDB来进行调试，那么本小节就来介绍一下如何通过GDB来调试Go程序。

GDB调试简介

GDB是FSF(自由软件基金会)发布的一个强大的类UNIX系统下的程序调试工具。使用GDB可以做如下事情：

1. 启动程序，可以按照开发者的自定义要求运行程序。
2. 可让被调试的程序在开发者设定的调置的断点处停住。（断点可以是条件表达式）
3. 当程序被停住时，可以检查此时程序中所发生的事。
4. 动态的改变当前程序的执行环境。

目前支持调试Go程序的GDB版本必须大于7.1。

编译Go程序的时候需要注意以下几点

1. 传递参数-ldflags "-s"，忽略debug的打印信息
2. 传递-gcflags "-N -l" 参数，这样可以忽略Go内部做的一些优化，聚合变量和函数等优化，这样对于GDB调试来说非常困难，所以在编译的时候加入这两个参数避免这些优化。

常用命令

GDB的一些常用命令如下所示

- list

  简写命令 `l` ，用来显示源代码，默认显示十行代码，后面可以带上参数显示的具体行，例如： `list 15` ，显示十行代码，其中第15行在显示的十行里面的中间，如下所示。

```
10	            time.Sleep(2 * time.Second)
11	            c <- i
12	        }
13	        close(c)
14	    }
15	
16	    func main() {
17	        msg := "Starting main"
18	        fmt.Println(msg)
19	        bus := make(chan int)
```

- break

  简写命令 `b`,用来设置断点，后面跟上参数设置断点的行数，例如 `b 10` 在第十行设置断点。

- delete 简写命令 `d`,用来删除断点，后面跟上断点设置的序号，这个序号可以通过 `info breakpoints` 获取相应的设置的断点序号，如下是显示的设置断点序号。

```
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x0000000000400dc3 in main.ma
breakpoint already hit 1 time
```

- backtrace

  简写命令 `bt`,用来打印执行的代码过程，如下所示：

```
#0  main.main () at /home/xiemengjun/gdb.go:23
#1  0x000000000040d61e in runtime.main () at /home/xiemengjun
#2  0x000000000040d6c1 in schedunlock () at /home/xiemengjun/
#3  0x0000000000000000 in ?? ()
```

- info

  info命令用来显示信息，后面有几种参数，我们常用的有如下几种：

  - `info locals`

    显示当前执行的程序中的变量值

  - `info breakpoints`

    显示当前设置的断点列表

  - `info goroutines`

显示当前执行的goroutine列表，如下代码所示,带*的表示当前执行的

```
* 1  running runtime.gosched
* 2  syscall runtime.entersyscall
  3  waiting runtime.gosched
  4 runnable runtime.gosched
```

- print

  简写命令 `p`，用来打印变量或者其他信息，后面跟上需要打印的变量名，当然还有一些很有用的函数$len()和$cap()，用来返回当前string、slices或者maps的长度和容量。

- whatis

  用来显示当前变量的类型，后面跟上变量名，例如 `whatis msg`,显示如下：

  ```
  type = struct string
  ```

- next

  简写命令 `n`,用来单步调试，跳到下一步，当有断点之后，可以输入 `n` 跳转到下一步继续执行

- coutinue

  简称命令 `c`，用来跳出当前断点处，后面可以跟参数N，跳过多少次断点

- set variable

  该命令用来改变运行过程中的变量值，格式如：`set variable <var>=<value>`

调试过程

我们通过下面这个代码来演示如何通过GDB来调试Go程序，下面是将要演示的代码：

```
package main

import (
    "fmt"
    "time"
)

func counting(c chan<- int) {
    for i := 0; i < 10; i++ {
        time.Sleep(2 * time.Second)
        c <- i
    }
    close(c)
}

func main() {
    msg := "Starting main"
    fmt.Println(msg)
    bus := make(chan int)
    msg = "starting a gofunc"
    go counting(bus)
    for count := range bus {
        fmt.Println("count:", count)
    }
}
```

编译文件，生成可执行文件gdbfile:

```
go build -gcflags "-N -l" gdbfile.go
```

通过gdb命令启动调试：

```
gdb gdbfile
```

启动之后首先看看这个程序是不是可以运行起来，只要输入 `run` 命令回车后程序就开始运行，程序正常的话可以看到程序输出如下，和我们在命令行直接执行程序输出是一样的：

```
(gdb) run
Starting program: /home/xiemengjun/gdbfile
Starting main
count: 0
count: 1
count: 2
count: 3
count: 4
count: 5
count: 6
count: 7
count: 8
count: 9
[LWP 2771 exited]
[Inferior 1 (process 2771) exited normally]
```

好了，现在我们已经知道怎么让程序跑起来了，接下来开始给代码设置断点：

```
(gdb) b 23
Breakpoint 1 at 0x400d8d: file /home/xiemengjun/gdbfile.go, line 23
(gdb) run
Starting program: /home/xiemengjun/gdbfile
Starting main
[New LWP 3284]
[Switching to LWP 3284]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23              fmt.Println("count:", count)
```

上面例子 `b 23` 表示在第23行设置了断点，之后输入 `run` 开始运行程序。现在程序在前面设置断点的地方停住了，我们需要查看断点相应上下文的源码，输入 `list` 就可以看到源码显示从当前停止行的前五行开始：

```
(gdb) list
18          fmt.Println(msg)
19          bus := make(chan int)
20          msg = "starting a gofunc"
21          go counting(bus)
22          for count := range bus {
23              fmt.Println("count:", count)
24          }
25      }
```

现在GDB在运行当前的程序的环境中已经保留了一些有用的调试信息，我们只需打印出相应的变量，查看相应变量的类型及值：

```
(gdb) info locals
count = 0
bus = 0xf840001a50
(gdb) p count
$1 = 0
(gdb) p bus
$2 = (chan int) 0xf840001a50
(gdb) whatis bus
type = chan int
```

接下来该让程序继续往下执行，请继续看下面的命令

```
(gdb) c
Continuing.
count: 0
[New LWP 3303]
[Switching to LWP 3303]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
(gdb) c
Continuing.
count: 1
[Switching to LWP 3302]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
```

每次输入 `c` 之后都会执行一次代码，又跳到下一次for循环，继续打印出来相应的信息。

设想目前需要改变上下文相关变量的信息，跳过一些过程，并继续执行下一步，得出修改后想要的结果：

```
(gdb) info locals
count = 2
bus = 0xf840001a50
(gdb) set variable count=9
(gdb) info locals
count = 9
bus = 0xf840001a50
(gdb) c
Continuing.
count: 9
[Switching to LWP 3302]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
```

最后稍微思考一下，前面整个程序运行的过程中到底创建了多少个goroutine，每个goroutine都在做什么：

```
(gdb) info goroutines
* 1 running runtime.gosched
* 2 syscall runtime.entersyscall
3 waiting runtime.gosched
4 runnable runtime.gosched
(gdb) goroutine 1 bt
#0 0x000000000040e33b in runtime.gosched () at /home/xiemengjun/go/
#1 0x0000000000403091 in runtime.chanrecv (c=void, ep=void, selecte
at /home/xiemengjun/go/src/pkg/runtime/chan.c:327
#2 0x000000000040316f in runtime.chanrecv2 (t=void, c=void)
at /home/xiemengjun/go/src/pkg/runtime/chan.c:420
#3 0x0000000000400d6f in main.main () at /home/xiemengjun/gdbfile.g
#4 0x000000000040d0c7 in runtime.main () at /home/xiemengjun/go/src
#5 0x000000000040d16a in schedunlock () at /home/xiemengjun/go/src/
#6 0x0000000000000000 in ?? ()
```

通过查看goroutines的命令我们可以清楚地了解goruntine内部是怎么执行的，每个函数的调用顺序已经明明白白地显示出来了。

小结

本小节我们介绍了GDB调试Go程序的一些基本命令，包括 `run` 、 `print` 、 `info` 、 `set variable` 、 `coutinue` 、 `list` 、 `break` 等经常用到的调试命令，通过上面的例子演示，我相信读者已经对于通过GDB调试Go程序有了基本的理解，如果你想获取更多的调试技巧请参考官方网站的GDB调试手册，还有GDB官方网站的手册。

links

- 目录
- 上一节: 错误处理
- 下一节: Go怎么写测试用例

11.3 Go怎么写测试用例

开发程序其中很重要的一点是测试，我们如何保证代码的质量，如何保证每个函数是可运行，运行结果是正确的，又如何保证写出来的代码性能是好的，我们知道单元测试的重点在于发现程序设计或实现的逻辑错误，使问题及早暴露，便于问题的定位解决，而性能测试的重点在于发现程序设计上的一些问题，让线上的程序能够在高并发的情况下还能保持稳定。本小节将带着这一连串的问题来讲解Go语言中如何来实现单元测试和性能测试。

Go语言中自带有一个轻量级的测试框架 `testing` 和自带的 `go test` 命令来实现单元测试和性能测试， `testing` 框架和其他语言中的测试框架类似，你可以基于这个框架写针对相应函数的测试用例，也可以基于该框架写相应的压力测试用例，那么接下来让我们一一来看一下怎么写。

如何编写测试用例

由于 `go test` 命令只能在一个相应的目录下执行所有文件，所以我们接下来新建一个项目目录 `gotest` ,这样我们所有的代码和测试代码都在这个目录下。

接下来我们在该目录下面创建两个文件：gotest.go和gotest_test.go

1. gotest.go:这个文件里面我们是创建了一个包，里面有一个函数实现了除法运算:

```
package gotest

import (
    "errors"
)

func Division(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("除数不能为0")
    }

    return a / b, nil
}
```

2. gotest_test.go:这是我们的单元测试文件，但是记住下面的这些原则：
    - 文件名必须是 `_test.go` 结尾的，这样在执行 `go test` 的时候才会执行到相应的代码
    - 你必须import `testing` 这个包
    - 所有的测试用例函数必须是 `Test` 开头
    - 测试用例会按照源代码中写的顺序依次执行

- 测试函数 `TestXxx()` 的参数是 `testing.T`，我们可以使用该类型来记录错误或者是测试状态
- 测试格式： `func TestXxx (t *testing.T)` ，`Xxx` 部分可以为任意的字母数字的组合，但是首字母不能是小写字母[a-z]，例如 `Testintdiv` 是错误的函数名。
- 函数中通过调用 `testing.T` 的 `Error` ， `Errorf` ， `FailNow` ， `Fatal` ， `FatalIf` 方法，说明测试不通过，调用 `Log` 方法用来记录测试的信息。

下面是我们的测试用例的代码：

package gotest

import (

```
"testing"
```

)

func Test_Division_1(t *testing.T) {

```
if i, e := Division(6, 2); i != 3 || e != nil { //try a u
    t.Error("除法函数测试没通过") // 如果不是如预期的那么就报错
} else {
    t.Log("第一个测试通过了") //记录一些你期望记录的信息
}
```

}

func Test_Division_2(t *testing.T) {

```
t.Error("就是不通过")
```

}

我们在项目目录下面执行 `go test` ,就会显示如下信息：

--- FAIL: Test_Division_2 (0.00 seconds)

```
gotest_test.go:16: 就是不通过
```

FAIL exit status 1 FAIL gotest 0.013s 从这个结果显示测试没有通过，因为在第二个测试函数中我们写死了测试不通过的代码 `t.Error` ，那么我们的第一个函数执行的情况怎么样呢？默认情况下执行 `go test` 是不会

显示测试通过的信息的，我们需要带上参数`go test -v`，这样就会显示如下信息：

=== RUN Test_Division_1 --- PASS: Test_Division_1 (0.00 seconds)

```
gotest_test.go:11: 第一个测试通过了
```

=== RUN Test_Division_2 --- FAIL: Test_Division_2 (0.00 seconds)

```
gotest_test.go:16: 就是不通过
```

FAIL exit status 1 FAIL gotest 0.012s 上面的输出详细的展示了这个测试的过程，我们看到测试函数1 `Test_Division_1` 测试通过，而测试函数2 `Test_Division_2` 测试失败了，最后得出结论测试不通过。接下来我们把测试函数2修改成如下代码：

func Test_Division_2(t *testing.T) {

```
if _, e := Division(6, 0); e == nil { //try a unit test o
    t.Error("Division did not work as expected.") // 如果不
} else {
    t.Log("one test passed.", e) //记录一些你期望记录的信息
}
```

}
然后我们执行`go test -v`，就显示如下信息，测试通过了：

=== RUN Test_Division_1 --- PASS: Test_Division_1 (0.00 seconds)

```
gotest_test.go:11: 第一个测试通过了
```

=== RUN Test_Division_2 --- PASS: Test_Division_2 (0.00 seconds)

```
gotest_test.go:20: one test passed. 除数不能为0
```

PASS ok gotest 0.013s

如何编写压力测试

压力测试用来检测函数(方法）的性能，和编写单元功能测试的方法类似,此处不再赘述，但需要注意以下几点：

- 压力测试用例必须遵循如下格式，其中XXX可以是任意字母数字的组合，但是首字母不能是小写字母

  ```
  func BenchmarkXXX(b *testing.B) { ... }
  ```

- `go test` 不会默认执行压力测试的函数，如果要执行压力测试需要带上参数 `-test.bench` ，语法: `-test.bench="test_name_regex"` ,例如 `go test -test.bench=".*"` 表示测试全部的压力测试函数
- 在压力测试用例中,请记得在循环体内使用 `testing.B.N` ,以使测试可以正常的运行
- 文件名也必须以 `_test.go` 结尾

下面我们新建一个压力测试文件webbench_test.go，代码如下所示：

```
package gotest

import (
    "testing"
)

func Benchmark_Division(b *testing.B) {
    for i := 0; i < b.N; i++ { //use b.N for looping
        Division(4, 5)
    }
}

func Benchmark_TimeConsumingFunction(b *testing.B) {
    b.StopTimer() //调用该函数停止压力测试的时间计数

    //做一些初始化的工作,例如读取文件数据,数据库连接之类的,
    //这样这些时间不影响我们测试函数本身的性能

    b.StartTimer() //重新开始时间
    for i := 0; i < b.N; i++ {
        Division(4, 5)
    }
}
```

我们执行命令 `go test -file webbench_test.go -test.bench=".*"` ，可以看到如下结果：

```
PASS
Benchmark_Division    500000000            7.76 ns/op
Benchmark_TimeConsumingFunction    500000000            7.80 ns/op
ok      gotest    9.364s
```

上面的结果显示我们没有执行任何 `TestXXX` 的单元测试函数，显示的结果只执行了压力测试函数，第一条显示了 `Benchmark_Division` 执行了500000000次，每次的执行平均时间是7.76纳秒，第二条显示了 `Benchmark_TimeConsumingFunction` 执行了500000000，每次的平均执行时间是7.80纳秒。最后一条显示总共的执行时间。

小结

通过上面对单元测试和压力测试的学习，我们可以看到 `testing` 包很轻量，编写单元测试和压力测试用例非常简单，配合内置的 `go test` 命令就可以非常方便的进行测试，这样在我们每次修改完代码,执行一下go test就可以简单的完成回归测试了。

links

- 目录
- 上一节: 使用GDB调试
- 下一节: 小结

11.4 小结

本章我们通过三个小节分别介绍了Go语言中如何处理错误，如何设计错误处理，然后第二小节介绍了如何通过GDB来调试程序，通过GDB我们可以单步调试、可以查看变量、修改变量、打印执行过程等，最后我们介绍了如何利用Go语言自带的轻量级框架 `testing` 来编写单元测试和压力测试，使用 `go test` 就可以方便的执行这些测试，使得我们将来代码升级修改之后很方便的进行回归测试。这一章也许对于你编写程序逻辑没有任何帮助，但是对于你编写出来的程序代码保持高质量是至关重要的，因为一个好的Web应用必定有良好的错误处理机制(错误提示的友好、可扩展性)、有好的单元测试和压力测试以保证上线之后代码能够保持良好的性能和按预期的运行。

links

- 目录
- 上一节: Go怎么写测试用例
- 下一节: 部署与维护

12 部署与维护

到目前为止，我们前面已经介绍了如何开发程序、调试程序以及测试程序，正如人们常说的：开发最后的10%需要花费90%的时间，所以这一章我们将强调这最后的10%部分，要真正成为让人信任并使用的优秀应用，需要考虑到一些细节，以上所说的10%就是指这些小细节。

本章我们将通过四个小节来介绍这些小细节的处理，第一小节介绍如何在生产服务上记录程序产生的日志，如何记录日志，第二小节介绍发生错误时我们的程序如何处理，如何保证尽量少的影响到用户的访问，第三小节介绍如何来部署Go的独立程序，由于目前Go程序还无法像C那样写成daemon，那么我们如何管理这样的进程程序后台运行呢？第四小节将介绍应用数据的备份和恢复，尽量保证应用在崩溃的情况能够保持数据的完整性。

目录

links

- 目录
- 上一章: 第十一章总结
- 下一节: 应用日志

12.1 应用日志

我们期望开发的Web应用程序能够把整个程序运行过程中出现的各种事件一一记录下来，Go语言中提供了一个简易的log包，我们使用该包可以方便的实现日志记录的功能，这些日志都是基于fmt包的打印再结合panic之类的函数来进行一般的打印、抛出错误处理。Go目前标准包只是包含了简单的功能，如果我们想把我们的应用日志保存到文件，然后又能够结合日志实现很多复杂的功能（编写过Java或者C++的读者应该都使用过log4j和log4cpp之类的日志工具），可以使用第三方开发的一个日志系统， `https://github.com/cihub/seelog` ，它实现了很强大的日志功能。接下来我们介绍如何通过该日志系统来实现我们应用的日志功能。

seelog介绍

seelog是用Go语言实现的一个日志系统，它提供了一些简单的函数来实现复杂的日志分配、过滤和格式化。主要有如下特性：

- XML的动态配置，可以不用重新编译程序而动态的加载配置信息
- 支持热更新，能够动态改变配置而不需要重启应用
- 支持多输出流，能够同时把日志输出到多种流中、例如文件流、网络流等
- 支持不同的日志输出
    - 命令行输出
    - 文件输出
    - 缓存输出
    - 支持log rotate
    - SMTP邮件

上面只列举了部分特性，seelog是一个特别强大的日志处理系统，详细的内容请参看官方wiki。接下来我将简要介绍一下如何在项目中使用它：

首先安装seelog

```
go get -u github.com/cihub/seelog
```

然后我们来看一个简单的例子：

```
package main

import log "github.com/cihub/seelog"

func main() {
    defer log.Flush()
    log.Info("Hello from Seelog!")
}
```

编译后运行如果出现了`Hello from seelog`，说明seelog日志系统已经成功安装并且可以正常运行了。

基于seelog的自定义日志处理

seelog支持自定义日志处理，下面是我基于它自定义的日志处理包的部分内容：

```
package logs

import (
    "errors"
    "fmt"
    seelog "github.com/cihub/seelog"
    "io"
)

var Logger seelog.LoggerInterface

func loadAppConfig() {
    appConfig := `
<seelog minlevel="warn">
    <outputs formatid="common">
        <rollingfile type="size" filename="/data/logs/roll.log" max
        <filter levels="critical">
            <file path="/data/logs/critical.log" formatid="critical
            <smtp formatid="criticalemail" senderaddress="astaxie@g
                <recipient address="xiemengjun@gmail.com"/>
            </smtp>
        </filter>
    </outputs>
    <formats>
        <format id="common" format="%Date/%Time [%LEV] %Msg%n" />
        <format id="critical" format="%File %FullPath %Func %Msg%n'
        <format id="criticalemail" format="Critical error on our se
    </formats>
</seelog>
`
    logger, err := seelog.LoggerFromConfigAsBytes([]byte(appConfig)
    if err != nil {
        fmt.Println(err)
        return
    }
    UseLogger(logger)
}

func init() {
    DisableLog()
    loadAppConfig()
}
```

```
// DisableLog disables all library log output
func DisableLog() {
    Logger = seelog.Disabled
}

// UseLogger uses a specified seelog.LoggerInterface to output libr
// Use this func if you are using Seelog logging system in your app
func UseLogger(newLogger seelog.LoggerInterface) {
    Logger = newLogger
}
```

上面主要实现了三个函数，

- `DisableLog`

  初始化全局变量Logger为seelog的禁用状态，主要为了防止Logger被多次初始化

- `loadAppConfig`

  根据配置文件初始化seelog的配置信息，这里我们把配置文件通过字符串读取设置好了，当然也可以通过读取XML文件。里面的配置说明如下：

  - seelog

    minlevel参数可选，如果被配置,高于或等于此级别的日志会被记录，同理maxlevel。

  - outputs

    输出信息的目的地，这里分成了两份数据，一份记录到log rotate文件里面。另一份设置了filter，如果这个错误级别是critical，那么将发送报警邮件。

  - formats

    定义了各种日志的格式

- `UseLogger`

  设置当前的日志器为相应的日志处理

上面我们定义了一个自定义的日志处理包，下面就是使用示例：

```
package main

import (
    "net/http"
    "project/logs"
    "project/configs"
    "project/routes"
)

func main() {
    addr, _ := configs.MainConfig.String("server", "addr")
    logs.Logger.Info("Start server at:%v", addr)
    err := http.ListenAndServe(addr, routes.NewMux())
    logs.Logger.Critical("Server err:%v", err)
}
```

发生错误发送邮件

上面的例子解释了如何设置发送邮件，我们通过如下的smtp配置用来发送邮件：

```
<smtp formatid="criticalemail" senderaddress="astaxie@gmail.com" se
    <recipient address="xiemengjun@gmail.com"/>
</smtp>
```

邮件的格式通过criticalemail配置，然后通过其他的配置发送邮件服务器的配置，通过recipient配置接收邮件的用户，如果有多个用户可以再添加一行。

要测试这个代码是否正常工作，可以在代码中增加类似下面的一个假消息。不过记住过后要把它删除，否则上线之后就会收到很多垃圾邮件。

```
logs.Logger.Critical("test Critical message")
```

现在，只要我们的应用在线上记录一个Critical的信息，你的邮箱就会收到一个Email，这样一旦线上的系统出现问题，你就能立马通过邮件获知，就能及时的进行处理。

使用应用日志

对于应用日志，每个人的应用场景可能会各不相同，有些人利用应用日志来做数据分析，有些人利用应用日志来做性能分析，有些人来做用户行为分析，还有些就是纯粹的记录，以方便应用出现问题的时候辅助查找问题。

举一个例子，我们需要跟踪用户尝试登陆系统的操作。这里会把成功与不成功的尝试都记录下来。记录成功的使用"Info"日志级别，而不成功的使用"warn"级别。如果想查找所有不成功的登陆，我们可以利用linux的grep之类的命令工具，如下：

```
# cat /data/logs/roll.log | grep "failed login"
2012-12-11 11:12:00 WARN : failed login attempt from 11.22.33.44 us
```

通过这种方式我们就可以很方便的查找相应的信息，这样有利于我们针对应用日志做一些统计和分析。另外我们还需要考虑日志的大小，对于一个高流量的Web应用来说，日志的增长是相当可怕的，所以我们在seelog的配置文件里面设置了logrotate，这样就能保证日志文件不会因为不断变大而导致我们的磁盘空间不够引起问题。

小结

通过上面对seelog系统及如何基于它进行自定义日志系统的学习，现在我们可以很轻松的随需构建一个合适的功能强大的日志处理系统了。日志处理系统为数据分析提供了可靠的数据源，比如通过对日志的分析，我们可以进一步优化系统，或者应用出现问题时方便查找定位问题，另外seelog也提供了日志分级功能，通过对minlevel的配置，我们可以很方便的设置测试或发布版本的输出消息级别。

links

- 目录
- 上一章: 部署与维护
- 下一节: 网站错误处理

12.2 网站错误处理

我们的Web应用一旦上线之后，那么各种错误出现的概率都有，Web应用日常运行中可能出现多种错误，具体如下所示：

- 数据库错误：指与访问数据库服务器或数据相关的错误。例如，以下可能出现的一些数据库错误。
    - 连接错误：这一类错误可能是数据库服务器网络断开、用户名密码不正确、或者数据库不存在。
    - 查询错误：使用的SQL非法导致错误，这样子SQL错误如果程序经过严格的测试应该可以避免。
    - 数据错误：数据库中的约束冲突，例如一个唯一字段中插入一条重复主键的值就会报错，但是如果你的应用程序在上线之前经过了严格的测试也是可以避免这类问题。
- 应用运行时错误：这类错误范围很广，涵盖了代码中出现的几乎所有错误。可能的应用错误的情况如下：
    - 文件系统和权限：应用读取不存在的文件，或者读取没有权限的文件、或者写入一个不允许写入的文件，这些都会导致一个错误。应用读取的文件如果格式不正确也会报错，例如配置文件应该是ini的配置格式，而设置成了json格式就会报错。
    - 第三方应用：如果我们的应用程序耦合了其他第三方接口程序，例如应用程序发表文章之后自动调用接发微博的接口，所以这个接口必须正常运行才能完成我们发表一篇文章的功能。
- HTTP错误：这些错误是根据用户的请求出现的错误，最常见的就是404错误。虽然可能会出现很多不同的错误，但其中比较常见的错误还有401未授权错误(需要认证才能访问的资源)、403禁止错误(不允许用户访问的资源)和503错误(程序内部出错)。
- 操作系统出错：这类错误都是由于应用程序上的操作系统出现错误引起的，主要有操作系统的资源被分配完了，导致死机，还有操作系统的磁盘满了，导致无法写入，这样就会引起很多错误。
- 网络出错：指两方面的错误，一方面是用户请求应用程序的时候出现网络断开，这样就导致连接中断，这种错误不会造成应用程序的崩溃，但是会影响用户访问的效果；另一方面是应用程序读取其他网络上的数据，其他网络断开会导致读取失败，这种需要对应用程序做有效的测试，能够避免这类问题出现的情况下程序崩溃。

错误处理的目标

在实现错误处理之前，我们必须明确错误处理想要达到的目标是什么，错误处理系统应该完成以下工作：

- 通知访问用户出现错误了：不论出现的是一个系统错误还是用户错误，用户都应当知道Web应用出了问题，用户的这次请求无法正确的完成了。例如，对于

用户的错误请求，我们显示一个统一的错误页面(404.html)。出现系统错误时，我们通过自定义的错误页面显示系统暂时不可用之类的错误页面(error.html)。
- 记录错误：系统出现错误，一般就是我们调用函数的时候返回err不为nil的情况，可以使用前面小节介绍的日志系统记录到日志文件。如果是一些致命错误，则通过邮件通知系统管理员。一般404之类的错误不需要发送邮件，只需要记录到日志系统。
- 回滚当前的请求操作：如果一个用户请求过程中出现了一个服务器错误，那么已完成的操作需要回滚。下面来看一个例子：一个系统将用户递交的表单保存到数据库，并将这个数据递交到一个第三方服务器，但是第三方服务器挂了，这就导致一个错误，那么先前存储到数据库的表单数据应该删除(应告知无效)，而且应该通知用户系统出现错误了。
- 保证现有程序可运行可服务：我们知道没有人能保证程序一定能够一直正常的运行着，万一哪一天程序崩溃了，那么我们就需要记录错误，然后立刻让程序重新运行起来，让程序继续提供服务，然后再通知系统管理员，通过日志等找出问题。

如何处理错误

错误处理其实我们已经在十一章第一小节里面有过介绍如何设计错误处理，这里我们再从一个例子详细的讲解一下，如何来处理不同的错误：

- 通知用户出现错误：

  通知用户在访问页面的时候我们可以有两种错误：404.html和error.html，下面分别显示了错误页面的源码：

```
<html lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; chars
    <title>找不到页面</title>
    <meta name="viewport" content="width=device-width, initia

</head>
<body>
<div class="container">
    <div class="row">
        <div class="span10">
            <div class="hero-unit">
                <h1>404!</h1>
                <p>{{.ErrorInfo}}</p>
            </div>
        </div><!--/span-->
    </div>
</div>
</body>
</html>
```

另一个源码：

```
<html lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; chars
    <title>系统错误页面</title>
    <meta name="viewport" content="width=device-width, initia

</head>
<body>
<div class="container">
    <div class="row">
        <div class="span10">
            <div class="hero-unit">
                <h1>系统暂时不可用!</h1>
                <p>{{.ErrorInfo}}</p>
            </div>
        </div><!--/span-->
    </div>
</div>
</body>
</html>
```

404的错误处理逻辑，如果是系统的错误也是类似的操作，同时我们看到在：

```
func (p *MyMux) ServeHTTP(w http.ResponseWriter, r *http.Requ
    if r.URL.Path == "/" {
        sayhelloName(w, r)
        return
    }
    NotFound404(w, r)
    return
}

func NotFound404(w http.ResponseWriter, r *http.Request) {
    log.Error("页面找不到")   //记录错误日志
    t, _ = t.ParseFiles("tmpl/404.html", nil)  //解析模板文件
    ErrorInfo := "文件找不到" //获取当前用户信息
    t.Execute(w, ErrorInfo)  //执行模板的merger操作
}

func SystemError(w http.ResponseWriter, r *http.Request) {
    log.Critical("系统错误")   //系统错误触发了Critical，那么不仅会
    t, _ = t.ParseFiles("tmpl/error.html", nil)  //解析模板文件
    ErrorInfo := "系统暂时不可用" //获取当前用户信息
    t.Execute(w, ErrorInfo)  //执行模板的merger操作
}
```

如何处理异常

我们知道在很多其他语言中有try...catch关键词，用来捕获异常情况，但是其实很多错误都是可以预期发生的，而不需要异常处理，应该当做错误来处理，这也是为什么Go语言采用了函数返回错误的设计，这些函数不会panic，例如如果一个文件找不到，os.Open返回一个错误，它不会panic；如果你向一个中断的网络连接写数据，net.Conn系列类型的Write函数返回一个错误，它们不会panic。这些状态在这样的程序里都是可以预期的。你知道这些操作可能会失败，因为设计者已经用返回错误清楚地表明了这一点。这就是上面所讲的可以预期发生的错误。

但是还有一种情况，有一些操作几乎不可能失败，而且在一些特定的情况下也没有办法返回错误，也无法继续执行，这样情况就应该panic。举个例子：如果一个程序计算x[j]，但是j越界了，这部分代码就会导致panic，像这样的一个不可预期严重错误就会引起panic，在默认情况下它会杀掉进程，它允许一个正在运行这部分代码的goroutine从发生错误的panic中恢复运行，发生panic之后，这部分代码后面的函数和代码都不会继续执行，这是Go特意这样设计的，因为要区别于错误和异常，panic其实就是异常处理。如下代码，我们期望通过uid来获取User中的username信息，但是如果uid越界了就会抛出异常，这个时候如果我们没有recover机制，进程就会被杀死，从而导致程序不可服务。因此为了程序的健壮性，在一些地方需要建立recover机制。

```
func GetUser(uid int) (username string) {
    defer func() {
        if x := recover(); x != nil {
            username = ""
        }
    }()

    username = User[uid]
    return
}
```

上面介绍了错误和异常的区别，那么我们在开发程序的时候如何来设计呢？规则很简单：如果你定义的函数有可能失败，它就应该返回一个错误。当我调用其他package的函数时，如果这个函数实现的很好，我不需要担心它会panic，除非有真正的异常情况发生，即使那样也不应该是我去处理它。而panic和recover是针对自己开发package里面实现的逻辑，针对一些特殊情况来设计。

小结

本小节总结了当我们的Web应用部署之后如何处理各种错误：网络错误、数据库错误、操作系统错误等，当错误发生时，我们的程序如何来正确处理：显示友好的出错界面、回滚操作、记录日志、通知管理员等操作，最后介绍了如何来正确处理错误和异常。一般的程序中错误和异常很容易混淆的，但是在Go中错误和异常是有明显的区分，所以告诉我们在程序设计中处理错误和异常应该遵循怎么样的原则。

links

- 目录
- 上一章: 应用日志
- 下一节: 应用部署

12.3 应用部署

程序开发完毕之后，我们现在要部署Web应用程序了，但是我们如何来部署这些应用程序呢？因为Go程序编译之后是一个可执行文件，编写过C程序的读者一定知道采用daemon就可以完美的实现程序后台持续运行，但是目前Go还无法完美的实现daemon，因此，针对Go的应用程序部署，我们可以利用第三方工具来管理，第三方的工具有很多，例如Supervisord、upstart、daemontools等，这小节我介绍目前自己系统中采用的工具Supervisord。

daemon

目前Go程序还不能实现daemon，详细的见这个Go语言的bug：`http://code.google.com/p/go/issues/detail?id=227`，大概的意思说很难从现有的使用的线程中fork一个出来，因为没有一种简单的方法来确保所有已经使用的线程的状态一致性问题。

但是我们可以看到很多网上的一些实现daemon的方法，例如下面两种方式：

- MarGo的一个实现思路，使用Commond来执行自身的应用，如果真想实现，那么推荐这种方案

```
d := flag.Bool("d", false, "Whether or not to launch in the b
if *d {
    cmd := exec.Command(os.Args[0],
        "-close-fds",
        "-addr", *addr,
        "-call", *call,
    )
    serr, err := cmd.StderrPipe()
    if err != nil {
        log.Fatalln(err)
    }
    err = cmd.Start()
    if err != nil {
        log.Fatalln(err)
    }
    s, err := ioutil.ReadAll(serr)
    s = bytes.TrimSpace(s)
    if bytes.HasPrefix(s, []byte("addr: ")) {
        fmt.Println(string(s))
        cmd.Process.Release()
    } else {
        log.Printf("unexpected response from MarGo: `%s` erro
        cmd.Process.Kill()
    }
}
```

- 另一种是利用syscall的方案，但是这个方案并不完善：

```
package main

import (
    "log"
    "os"
    "syscall"
)

func daemon(nochdir, noclose int) int {
    var ret, ret2 uintptr
    var err uintptr

    darwin := syscall.OS == "darwin"

    // already a daemon
    if syscall.Getppid() == 1 {
        return 0
    }

    // fork off the parent process
    ret, ret2, err = syscall.RawSyscall(syscall.SYS_FORK, 0,
    if err != 0 {
```

```
        return -1
    }

    // failure
    if ret2 < 0 {
        os.Exit(-1)
    }

    // handle exception for darwin
    if darwin && ret2 == 1 {
        ret = 0
    }

    // if we got a good PID, then we call exit the parent pro
    if ret > 0 {
        os.Exit(0)
    }

    /* Change the file mode mask */
    _ = syscall.Umask(0)

    // create a new SID for the child process
    s_ret, s_errno := syscall.Setsid()
    if s_errno != 0 {
        log.Printf("Error: syscall.Setsid errno: %d", s_errno
    }
    if s_ret < 0 {
        return -1
    }

    if nochdir == 0 {
        os.Chdir("/")
    }

    if noclose == 0 {
        f, e := os.OpenFile("/dev/null", os.O_RDWR, 0)
        if e == nil {
            fd := f.Fd()
            syscall.Dup2(fd, os.Stdin.Fd())
            syscall.Dup2(fd, os.Stdout.Fd())
            syscall.Dup2(fd, os.Stderr.Fd())
        }
    }

    return 0
}
```

上面提出了两种实现Go的daemon方案，但是我还是不推荐大家这样去实现，因为官方还没有正式的宣布支持daemon，当然第一种方案目前来看是比较可行的，而且目前开源库skynet也在采用这个方案做daemon。

Supervisord

上面已经介绍了Go目前是有两种方案来实现他的daemon，但是官方本身还不支持这一块，所以还是建议大家采用第三方成熟工具来管理我们的应用程序，这里我给大家介绍一款目前使用比较广泛的进程管理软件：Supervisord。Supervisord是用Python实现的一款非常实用的进程管理工具。supervisord会帮你把管理的应用程序转成daemon程序，而且可以方便的通过命令开启、关闭、重启等操作，而且它管理的进程一旦崩溃会自动重启，这样就可以保证程序执行中断后的情况下有自我修复的功能。

> 我前面在应用中踩过一个坑，就是因为所有的应用程序都是由Supervisord父进程生出来的，那么当你修改了操作系统的文件描述符之后，别忘记重启Supervisord，光重启下面的应用程序没用。当初我就是系统安装好之后就先装了Supervisord，然后开始部署程序，修改文件描述符，重启程序，以为文件描述符已经是100000了，其实Supervisord这个时候还是默认的1024个，导致他管理的进程所有的描述符也是1024.开放之后压力一上来系统就开始报文件描述符用光了，查了很久才找到这个坑。

Supervisord安装

Supervisord可以通过 `sudo easy_install supervisor` 安装，当然也可以通过Supervisord官网下载后解压并转到源码所在的文件夹下执行 `setup.py install` 来安装。

- 使用easy_install必须安装setuptools

  打开 `http://pypi.python.org/pypi/setuptools#files` ，根据你系统的python的版本下载相应的文件，然后执行 `sh setuptoolsxxxx.egg` ，这样就可以使用easy_install命令来安装Supervisord。

Supervisord配置

Supervisord默认的配置文件路径为/etc/supervisord.conf，通过文本编辑器修改这个文件，下面是一个示例的配置文件：

```
;/etc/supervisord.conf
[unix_http_server]
file = /var/run/supervisord.sock
chmod = 0777
chown= root:root

[inet_http_server]
# Web管理界面设定
port=9001
username = admin
password = yourpassword

[supervisorctl]
; 必须和'unix_http_server'里面的设定匹配
serverurl = unix:///var/run/supervisord.sock

[supervisord]
logfile=/var/log/supervisord/supervisord.log ; (main log file;defau
logfile_maxbytes=50MB       ; (max main logfile bytes b4 rotation;d
logfile_backups=10          ; (num of main logfile rotation backups
loglevel=info               ; (log level;default info; others: debu
pidfile=/var/run/supervisord.pid ; (supervisord pidfile;default sup
nodaemon=true              ; (start in foreground if true;default f
minfds=1024                 ; (min. avail startup file descriptors;
minprocs=200                ; (min. avail process descriptors;defau
user=root                  ; (default is current user, required if r
childlogdir=/var/log/supervisord/            ; ('AUTO' child log di

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_mair

; 管理的单个进程的配置，可以添加多个program
[program:blogdemon]
command=/data/blog/blogdemon
autostart = true
startsecs = 5
user = root
redirect_stderr = true
stdout_logfile = /var/log/supervisord/blogdemon.log
```

Supervisord管理

Supervisord安装完成后有两个可用的命令行supervisor和supervisorctl，命令使用解释如下：

- supervisord，初始启动Supervisord，启动、管理配置中设置的进程。
- supervisorctl stop programxxx，停止某一个进程(programxxx)，programxxx为[program:blogdemon]里配置的值，这个示例就是blogdemon。
- supervisorctl start programxxx，启动某个进程

- supervisorctl restart programxxx，重启某个进程
- supervisorctl stop all，停止全部进程，注：start、restart、stop都不会载入最新的配置文件。
- supervisorctl reload，载入最新的配置文件，并按新的配置启动、管理所有进程。

小结

这小节我们介绍了Go如何实现daemon化，但是由于目前Go的daemon实现的不足，需要依靠第三方工具来实现应用程序的daemon管理的方式，所以在这里介绍了一个用python写的进程管理工具Supervisord，通过Supervisord可以很方便的把我们的Go应用程序管理起来。

links

- 目录
- 上一章: 网站错误处理
- 下一节: 备份和恢复

12.4 备份和恢复

这小节我们要讨论应用程序管理的另一个方面：生产服务器上数据的备份和恢复。我们经常会遇到生产服务器的网络断了、硬盘坏了、操作系统崩溃、或者数据库不可用了等各种异常情况，所以维护人员需要对生产服务器上的应用和数据做好异地灾备，冷备热备的准备。在接下来的介绍中，讲解了如何备份应用、如何备份/恢复Mysql数据库和redis数据库。

应用备份

在大多数集群环境下，Web应用程序基本不需要备份，因为这个其实就是一个代码副本，我们在本地开发环境中，或者版本控制系统中已经保持这些代码。但是很多时候，一些开发的站点需要用户来上传文件，那么我们需要对这些用户上传的文件进行备份。目前其实有一种合适的做法就是把和网站相关的需要存储的文件存储到云储存，这样即使系统崩溃，只要我们的文件还在云存储上，至少数据不会丢失。

如果我们没有采用云储存的情况下，如何做到网站的备份呢？这里我们介绍一个文件同步工具rsync：rsync能够实现网站的备份，不同系统的文件的同步，如果是windows的话，需要windows版本cwrsync。

rsync安装

rysnc的官方网站：http://rsync.samba.org/ 可以从上面获取最新版本的源码。当然，因为rsync是一款非常有用的软件，所以很多Linux的发行版本都将它收录在内了。

软件包安装

```
# sudo apt-get  install  rsync  注：在debian、ubuntu 等在线安装方法；
# yum install rsync     注：Fedora、Redhat、CentOS 等在线安装方法；
# rpm -ivh rsync        注：Fedora、Redhat、CentOS 等rpm包安装方法；
```

其它Linux发行版，请用相应的软件包管理方法来安装。源码包安装

```
tar xvf  rsync-xxx.tar.gz
cd rsync-xxx
./configure --prefix=/usr  ;make ;make install   注：在用源码包编译安装
```

rsync配置

rsync主要有以下三个配置文件rsyncd.conf(主配置文件)、rsyncd.secrets(密码文件)、rsyncd.motd(rysnc服务器信息)。

关于这几个文件的配置大家可以参考官方网站或者其他介绍rsync的网站，下面介绍服务器端和客户端如何开启

- 服务端开启：

```
#/usr/bin/rsync --daemon  --config=/etc/rsyncd.conf
```

  --daemon参数方式，是让rsync以服务器模式运行。把rsync加入开机启动

```
echo 'rsync --daemon' >> /etc/rc.d/rc.local
```

  设置rsync密码

```
echo '你的用户名:你的密码' > /etc/rsyncd.secrets
chmod 600 /etc/rsyncd.secrets
```

- 客户端同步：

  客户端可以通过如下命令同步服务器上的文件：

```
rsync -avzP  --delete  --password-file=rsyncd.secrets   用户名
```

  这条命令，简要的说明一下几个要点：

  1. -avzP是啥，读者可以使用--help查看
  2. --delete 是为了比如A上删除了一个文件，同步的时候，B会自动删除相对应的文件
  3. --password-file 客户端中/etc/rsyncd.secrets设置的密码，要和服务端的 /etc/rsyncd.secrets 中的密码一样，这样cron运行的时候，就不需要密码了
  4. 这条命令中的"用户名"为服务端的 /etc/rsyncd.secrets中的用户名
  5. 这条命令中的 192.168.145.5 为服务端的IP地址
  6. ::www，注意是2个 : 号，www为服务端的配置文件 /etc/rsyncd.conf 中的[www]，意思是根据服务端上的/etc/rsyncd.conf来同步其中的[www]段内容，一个 : 号的时候，用于不根据配置文件，直接同步指定目录。

     为了让同步实时性，可以设置crontab，保持rsync每分钟同步，当然用户也可以根据文件的重要程度设置不同的同步频率。

MySQL备份

应用数据库目前还是MySQL为主流，目前MySQL的备份有两种方式：热备份和冷备份，热备份目前主要是采用master/slave方式（master/slave方式的同步目前主要用于数据库读写分离，也可以用于热备份数据），关于如何配置这方面的资料，大家可以找到很多。冷备份的话就是数据有一定的延迟，但是可以保证该时间段之前的数据完整，例如有些时候可能我们的误操作引起了数据的丢失，那么master/slave模式是无法找回丢失数据的，但是通过冷备份可以部分恢复数据。

冷备份一般使用shell脚本来实现定时备份数据库，然后通过上面介绍rsync同步非本地机房的一台服务器。

下面这个是定时备份mysql的备份脚本，我们使用了mysqldump程序，这个命令可以把数据库导出到一个文件中。

```
#!/bin/bash

# 以下配置信息请自己修改
mysql_user="USER" #MySQL备份用户
mysql_password="PASSWORD" #MySQL备份用户的密码
mysql_host="localhost"
mysql_port="3306"
mysql_charset="utf8" #MySQL编码
backup_db_arr=("db1" "db2") #要备份的数据库名称，多个用空格分开隔开 如("d
backup_location=/var/www/mysql  #备份数据存放位置，末尾请不要带"/",此项可
expire_backup_delete="ON" #是否开启过期备份删除 ON为开启 OFF为关闭
expire_days=3 #过期时间天数 默认为三天，此项只有在expire_backup_delete开

# 本行开始以下不需要修改
backup_time=`date +%Y%m%d%H%M`  #定义备份详细时间
backup_Ymd=`date +%Y-%m-%d` #定义备份目录中的年月日时间
backup_3ago=`date -d '3 days ago' +%Y-%m-%d` #3天之前的日期
backup_dir=$backup_location/$backup_Ymd  #备份文件夹全路径
welcome_msg="Welcome to use MySQL backup tools!" #欢迎语

# 判断MYSQL是否启动,mysql没有启动则备份退出
mysql_ps=`ps -ef |grep mysql |wc -l`
mysql_listen=`netstat -an |grep LISTEN |grep $mysql_port|wc -l`
if [ [$mysql_ps == 0] -o [$mysql_listen == 0] ]; then
        echo "ERROR:MySQL is not running! backup stop!"
        exit
else
        echo $welcome_msg
fi

# 连接到mysql数据库，无法连接则备份退出
mysql -h$mysql_host -P$mysql_port -u$mysql_user -p$mysql_password <
use mysql;
select host,user from user where user='root' and host='localhost';
exit
end

flag=`echo $?`
if [ $flag != "0" ]; then
```

```
        echo "ERROR:Can't connect mysql server! backup stop!"
        exit
else
        echo "MySQL connect ok! Please wait......"
        # 判断有没有定义备份的数据库，如果定义则开始备份，否则退出备份
        if [ "$backup_db_arr" != "" ];then
                #dbnames=$(cut -d ',' -f1-5 $backup_database)
                #echo "arr is (${backup_db_arr[@]})"
                for dbname in ${backup_db_arr[@]}
                do
                        echo "database $dbname backup start..."
                        `mkdir -p $backup_dir`
                        `mysqldump -h$mysql_host -P$mysql_port -u$m
                        flag=`echo $?`
                        if [ $flag == "0" ];then
                                echo "database $dbname success back
                        else
                                echo "database $dbname backup fail
                        fi

                done
        else
                echo "ERROR:No database to backup! backup stop"
                exit
        fi
        # 如果开启了删除过期备份，则进行删除操作
        if [ "$expire_backup_delete" == "ON" -a  "$backup_location"
                 #`find $backup_location/ -type d -o -type f -ctime
                 `find $backup_location/ -type d -mtime +$expire_da
                 echo "Expired backup data delete complete!"
        fi
        echo "All database backup success! Thank you!"
        exit
fi
```

修改shell脚本的属性：

```
chmod 600 /root/mysql_backup.sh
chmod +x /root/mysql_backup.sh
```

设置好属性之后，把命令加入crontab，我们设置了每天00:00定时自动备份，然后把备份的脚本目录/var/www/mysql设置为rsync同步目录。

```
00 00 * * * /root/mysql_backup.sh
```

MySQL恢复

前面介绍MySQL备份分为热备份和冷备份，热备份主要的目的是为了能够实时的恢复，例如应用服务器出现了硬盘故障，那么我们可以通过修改配置文件把数据库的读取和写入改成slave，这样就可以尽量少时间的中断服务。

但是有时候我们需要通过冷备份的SQL来进行数据恢复，既然有了数据库的备份，就可以通过命令导入：

```
mysql -u username -p databse < backup.sql
```

可以看到，导出和导入数据库数据都是相当简单，不过如果还需要管理权限，或者其他的一些字符集的设置的话，可能会稍微复杂一些，但是这些都是可以通过一些命令来完成的。

redis备份

redis是目前我们使用最多的NoSQL，它的备份也分为两种：热备份和冷备份，redis也支持master/slave模式，所以我们的热备份可以通过这种方式实现，相应的配置大家可以参考官方的文档配置，相当的简单。我们这里介绍冷备份的方式：redis其实会定时的把内存里面的缓存数据保存到数据库文件里面，我们备份只要备份相应的文件就可以，就是利用前面介绍的rsync备份到非本地机房就可以实现。

redis恢复

redis的恢复分为热备份恢复和冷备份恢复，热备份恢复的目的和方法同MySQL的恢复一样，只要修改应用的相应的数据库连接即可。

但是有时候我们需要根据冷备份来恢复数据，redis的冷备份恢复其实就是只要把保存的数据库文件copy到redis的工作目录，然后启动redis就可以了，redis在启动的时候会自动加载数据库文件到内存中，启动的速度根据数据库的文件大小来决定。

小结

本小节介绍了我们的应用部分的备份和恢复，即如何做好灾备，包括文件的备份、数据库的备份。同时也介绍了使用rsync同步不同系统的文件，MySQL数据库和redis数据库的备份和恢复，希望通过本小节的介绍，能够给作为开发的你对于线上产品的灾备方案提供一个参考方案。

links

- 目录
- 上一章: 应用部署
- 下一节: 小结

12.5 小结

本章讨论了如何部署和维护我们开发的Web应用相关的一些话题。这些内容非常重要，要创建一个能够基于最小维护平滑运行的应用，必须考虑这些问题。

具体而言，本章讨论的内容包括：

- 创建一个强健的日志系统，可以在出现问题时记录错误并且通知系统管理员
- 处理运行时可能出现的错误，包括记录日志，并如何友好的显示给用户系统出现了问题
- 处理404错误，告诉用户请求的页面找不到
- 将应用部署到一个生产环境中(包括如何部署更新)
- 如何让部署的应用程序具有高可用
- 备份和恢复文件以及数据库

读完本章内容后，对于从头开始开发一个Web应用需要考虑那些问题，你应该已经有了全面的了解。本章内容将有助于你在实际环境中管理前面各章介绍开发的代码。

links

- 目录
- 上一章: 备份和恢复
- 下一节: 如何设计一个Web框架

13 如何设计一个Web框架

前面十二章介绍了如何通过Go来开发Web应用，介绍了很多基础知识、开发工具和开发技巧，那么我们这一章通过这些知识来实现一个简易的Web框架。通过Go语言来实现一个完整的框架设计，这框架中主要内容有第一小节介绍的Web框架的结构规划，例如采用MVC模式来进行开发，程序的执行流程设计等内容；第二小节介绍框架的第一个功能：路由，如何让访问的URL映射到相应的处理逻辑；第三小节介绍处理逻辑，如何设计一个公共的controller，对象继承之后处理函数中如何处理response和request；第四小节介绍如何框架的一些辅助功能，例如日志处理、配置信息等；第五小节介绍如何基于Web框架实现一个博客，包括博文的发表、修改、删除、显示列表等操作。

通过这么一个完整的项目例子，我期望能够让读者了解如何开发Web应用，如何搭建自己的目录结构，如何实现路由，如何实现MVC模式等各方面的开发内容。在框架盛行的今天，MVC也不再是神话。经常听到很多程序员讨论哪个框架好，哪个框架不好， 其实框架只是工具，没有好与不好，只有适合与不适合，适合自己的就是最好的，所以教会大家自己动手写框架，那么不同的需求都可以用自己的思路去实现。

目录

links

- 目录
- 上一章: 第十二章总结
- 下一节: 项目规划

13.1 项目规划

做任何事情都需要做好规划，那么我们在开发博客系统之前，同样需要做好项目的规划，如何设置目录结构，如何理解整个项目的流程图，当我们理解了应用的执行过程，那么接下来的设计编码就会变得相对容易了

gopath以及项目设置

假设指定gopath是文件系统的普通目录名，当然我们可以随便设置一个目录名，然后将其路径存入GOPATH。前面介绍过GOPATH可以是多个目录：在window系统设置环境变量；在linux/MacOS系统只要输入终端命令 `export gopath=/home/astaxie/gopath` ，但是必须保证gopath这个代码目录下面有三个目录pkg、bin、src。新建项目的源码放在src目录下面，现在暂定我们的博客目录叫做beeblog，下面是在window下的环境变量和目录结构的截图：

图13.1 环境变量GOPATH设置

图13.2 工作目录在$gopath/src下

应用程序流程图

博客系统是基于模型-视图-控制器这一设计模式的。MVC是一种将应用程序的逻辑层和表现层进行分离的结构方式。在实践中，由于表现层从Go中分离了出来，所以它允许你的网页中只包含很少的脚本。

- 模型 (Model) 代表数据结构。通常来说，模型类将包含取出、插入、更新数据库资料等这些功能。
- 视图 (View) 是展示给用户的信息的结构及样式。一个视图通常是一个网页，但是在Go中，一个视图也可以是一个页面片段，如页头、页尾。它还可以是一个RSS 页面，或其它类型的“页面”，Go实现的template包已经很好的实现了View层中的部分功能。
- 控制器 (Controller) 是模型、视图以及其他任何处理HTTP请求所必须的资源之间的中介，并生成网页。

下图显示了项目设计中框架的数据流是如何贯穿整个系统:

图13.3 框架的数据流

1. main.go作为应用入口，初始化一些运行博客所需要的基本资源，配置信息，监听端口。

2. 路由功能检查HTTP请求，根据URL以及method来确定谁(控制层)来处理请求的转发资源。
3. 如果缓存文件存在，它将绕过通常的流程执行，被直接发送给浏览器。
4. 安全检测：应用程序控制器调用之前，HTTP请求和任一用户提交的数据将被过滤。
5. 控制器装载模型、核心库、辅助函数，以及任何处理特定请求所需的其它资源，控制器主要负责处理业务逻辑。
6. 输出视图层中渲染好的即将发送到Web浏览器中的内容。如果开启缓存，视图首先被缓存，将用于以后的常规请求。

目录结构

根据上面的应用程序流程设计，博客的目录结构设计如下：

```
|——main.go          入口文件
|——conf             配置文件和处理模块
|——controllers      控制器入口
|——models           数据库处理模块
|——utils            辅助函数库
|——static           静态文件目录
|——views            视图库
```

框架设计

为了实现博客的快速搭建，打算基于上面的流程设计开发一个最小化的框架，框架包括路由功能、支持REST的控制器、自动化的模板渲染，日志系统、配置管理等。

总结

本小节介绍了博客系统从设置GOPATH到目录建立这样的基础信息，也简单介绍了框架结构采用的MVC模式，博客系统中数据流的执行流程，最后通过这些流程设计了博客系统的目录结构，至此，我们基本完成一个框架的搭建，接下来的几个小节我们将会逐个实现。

links

- 目录
- 上一章: 构建博客系统
- 下一节: 自定义路由器设计

13.2 自定义路由器设计

HTTP路由

HTTP路由组件负责将HTTP请求交到对应的函数处理(或者是一个struct的方法)，如前面小节所描述的结构图，路由在框架中相当于一个事件处理器，而这个事件包括：

- 用户请求的路径(path)(例如:/user/123,/article/123)，当然还有查询串信息(例如?id=11)
- HTTP的请求方法(method)(GET、POST、PUT、DELETE、PATCH等)

路由器就是根据用户请求的事件信息转发到相应的处理函数(控制层)。

默认的路由实现

在3.4小节有过介绍Go的http包的详解，里面介绍了Go的http包如何设计和实现路由，这里继续以一个例子来说明：

```
func fooHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
}

http.HandleFunc("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

上面的例子调用了http默认的DefaultServeMux来添加路由，需要提供两个参数，第一个参数是希望用户访问此资源的URL路径(保存在r.URL.Path)，第二参数是即将要执行的函数，以提供用户访问的资源。路由的思路主要集中在两点：

- 添加路由信息
- 根据用户请求转发到要执行的函数

Go默认的路由添加是通过函数 `http.Handle` 和 `http.HandleFunc` 等来添加，底层都是调用了 `DefaultServeMux.Handle(pattern string, handler Handler)` ,这个函数会把路由信息存储在一个map信息中 `map[string]muxEntry` ，这就解决了上面说的第一点。

Go监听端口，然后接收到tcp连接会扔给Handler来处理，上面的例子默认nil即为 `http.DefaultServeMux` ，通过 `DefaultServeMux.ServeHTTP` 函数来进行调度，遍历之前存储的map路由信息，和用户访问的URL进行匹配，以查询对应注册的处理函数，这样就实现了上面所说的第二点。

```
for k, v := range mux.m {
    if !pathMatch(k, path) {
        continue
    }
    if h == nil || len(k) > n {
        n = len(k)
        h = v.h
    }
}
```

beego框架路由实现

目前几乎所有的Web应用路由实现都是基于http默认的路由器，但是Go自带的路由器有几个限制：

- 不支持参数设定，例如/user/:uid 这种泛类型匹配
- 无法很好的支持REST模式，无法限制访问的方法，例如上面的例子中，用户访问/foo，可以用GET、POST、DELETE、HEAD等方式访问
- 一般网站的路由规则太多了，编写繁琐。我前面自己开发了一个API应用，路由规则有三十几条，这种路由多了之后其实可以进一步简化，通过struct的方法进行一种简化

beego框架的路由器基于上面的几点限制考虑设计了一种REST方式的路由实现，路由设计也是基于上面Go默认设计的两点来考虑：存储路由和转发路由

存储路由

针对前面所说的限制点，我们首先要解决参数支持就需要用到正则，第二和第三点我们通过一种变通的方法来解决，REST的方法对应到struct的方法中去，然后路由到struct而不是函数，这样在转发路由的时候就可以根据method来执行不同的方法。

根据上面的思路，我们设计了两个数据类型controllerInfo(保存路径和对应的struct，这里是一个reflect.Type类型)和ControllerRegistor(routers是一个slice用来保存用户添加的路由信息，以及beego框架的应用信息)

```
type controllerInfo struct {
    regex          *regexp.Regexp
    params         map[int]string
    controllerType reflect.Type
}

type ControllerRegistor struct {
    routers     []*controllerInfo
    Application *App
}
```

ControllerRegistor对外的接口函数有

```
func (p *ControllerRegistor) Add(pattern string, c ControllerInterf
```

详细的实现如下所示：

```
func (p *ControllerRegistor) Add(pattern string, c ControllerInterf
    parts := strings.Split(pattern, "/")

    j := 0
    params := make(map[int]string)
    for i, part := range parts {
        if strings.HasPrefix(part, ":") {
            expr := "([^/]+)"

            //a user may choose to override the defult expression
            // similar to expressjs: ‘/user/:id([0-9]+)’

            if index := strings.Index(part, "("); index != -1 {
                expr = part[index:]
                part = part[:index]
            }
            params[j] = part
            parts[i] = expr
            j++
        }
    }

    //recreate the url pattern, with parameters replaced
    //by regular expressions. then compile the regex

    pattern = strings.Join(parts, "/")
    regex, regexErr := regexp.Compile(pattern)
    if regexErr != nil {

        //TODO add error handling here to avoid panic
        panic(regexErr)
        return
    }

    //now create the Route
    t := reflect.Indirect(reflect.ValueOf(c)).Type()
    route := &controllerInfo{}
    route.regex = regex
    route.params = params
    route.controllerType = t

    p.routers = append(p.routers, route)

}
```

静态路由实现

上面我们实现的动态路由的实现，Go的http包默认支持静态文件处理FileServer，由于我们实现了自定义的路由器，那么静态文件也需要自己设定，beego的静态文件夹路径保存在全局变量StaticDir中，StaticDir是一个map类型，实现如下：

```
func (app *App) SetStaticPath(url string, path string) *App {
    StaticDir[url] = path
    return app
}
```

应用中设置静态路径可以使用如下方式实现：

```
beego.SetStaticPath("/img","/static/img")
```

转发路由

转发路由是基于ControllerRegistor里的路由信息来进行转发的，详细的实现如下代码所示：

```
// AutoRoute
func (p *ControllerRegistor) ServeHTTP(w http.ResponseWriter, r *ht
    defer func() {
        if err := recover(); err != nil {
            if !RecoverPanic {
                // go back to panic
                panic(err)
            } else {
                Critical("Handler crashed with error", err)
                for i := 1; ; i += 1 {
                    _, file, line, ok := runtime.Caller(i)
                    if !ok {
                        break
                    }
                    Critical(file, line)
                }
            }
        }
    }()
    var started bool
    for prefix, staticDir := range StaticDir {
        if strings.HasPrefix(r.URL.Path, prefix) {
            file := staticDir + r.URL.Path[len(prefix):]
            http.ServeFile(w, r, file)
            started = true
            return
        }
    }
    requestPath := r.URL.Path
```

```
    //find a matching Route
    for _, route := range p.routers {

        //check if Route pattern matches url
        if !route.regex.MatchString(requestPath) {
            continue
        }

        //get submatches (params)
        matches := route.regex.FindStringSubmatch(requestPath)

        //double check that the Route matches the URL pattern.
        if len(matches[0]) != len(requestPath) {
            continue
        }

        params := make(map[string]string)
        if len(route.params) > 0 {
            //add url parameters to the query param map
            values := r.URL.Query()
            for i, match := range matches[1:] {
                values.Add(route.params[i], match)
                params[route.params[i]] = match
            }

            //reassemble query params and add to RawQuery
            r.URL.RawQuery = url.Values(values).Encode() + "&" + r.
            //r.URL.RawQuery = url.Values(values).Encode()
        }
        //Invoke the request handler
        vc := reflect.New(route.controllerType)
        init := vc.MethodByName("Init")
        in := make([]reflect.Value, 2)
        ct := &Context{ResponseWriter: w, Request: r, Params: paran
        in[0] = reflect.ValueOf(ct)
        in[1] = reflect.ValueOf(route.controllerType.Name())
        init.Call(in)
        in = make([]reflect.Value, 0)
        method := vc.MethodByName("Prepare")
        method.Call(in)
        if r.Method == "GET" {
            method = vc.MethodByName("Get")
            method.Call(in)
        } else if r.Method == "POST" {
            method = vc.MethodByName("Post")
            method.Call(in)
        } else if r.Method == "HEAD" {
            method = vc.MethodByName("Head")
            method.Call(in)
        } else if r.Method == "DELETE" {
            method = vc.MethodByName("Delete")
            method.Call(in)
        } else if r.Method == "PUT" {
```

```
            method = vc.MethodByName("Put")
            method.Call(in)
        } else if r.Method == "PATCH" {
            method = vc.MethodByName("Patch")
            method.Call(in)
        } else if r.Method == "OPTIONS" {
            method = vc.MethodByName("Options")
            method.Call(in)
        }
        if AutoRender {
            method = vc.MethodByName("Render")
            method.Call(in)
        }
        method = vc.MethodByName("Finish")
        method.Call(in)
        started = true
        break
    }

    //if no matches to url, throw a not found exception
    if started == false {
        http.NotFound(w, r)
    }
}
```

使用入门

基于这样的路由设计之后就可以解决前面所说的三个限制点，使用的方式如下所示：

基本的使用注册路由：

```
beego.BeeApp.RegisterController("/", &controllers.MainController{})
```

参数注册：

```
beego.BeeApp.RegisterController("/:param", &controllers.UserContro
```

正则匹配：

```
beego.BeeApp.RegisterController("/users/:uid([0-9]+)", &controllers
```

links

- 目录
- 上一章: 项目规划
- 下一节: controller设计

13.3 controller设计

传统的MVC框架大多数是基于Action设计的后缀式映射，然而，现在Web流行REST风格的架构。尽管使用Filter或者rewrite能够通过URL重写实现REST风格的URL，但是为什么不直接设计一个全新的REST风格的 MVC框架呢？本小节就是基于这种思路来讲述如何从头设计一个基于REST风格的MVC框架中的controller，最大限度地简化Web应用的开发，甚至编写一行代码就可以实现“Hello, world”。

controller作用

MVC设计模式是目前Web应用开发中最常见的架构模式，通过分离 Model（模型）、View（视图）和 Controller（控制器），可以更容易实现易于扩展的用户界面(UI)。Model指后台返回的数据；View指需要渲染的页面，通常是模板页面，渲染后的内容通常是HTML；Controller指Web开发人员编写的处理不同URL的控制器，如前面小节讲述的路由就是URL请求转发到控制器的过程，controller在整个的MVC框架中起到了一个核心的作用，负责处理业务逻辑，因此控制器是整个框架中必不可少的一部分，Model和View对于有些业务需求是可以不写的，例如没有数据处理的逻辑处理，没有页面输出的302调整之类的就不需要Model和View，但是controller这一环节是必不可少的。

beego的REST设计

前面小节介绍了路由实现了注册struct的功能，而struct中实现了REST方式，因此我们需要设计一个用于逻辑处理controller的基类，这里主要设计了两个类型，一个struct、一个interface

```
type Controller struct {
    Ct        *Context
    Tpl       *template.Template
    Data      map[interface{}]interface{}
    ChildName string
    TplNames  string
    Layout    []string
    TplExt    string
}

type ControllerInterface interface {
    Init(ct *Context, cn string)    //初始化上下文和子类名称
    Prepare()                       //开始执行之前的一些处理
    Get()                           //method=GET的处理
    Post()                          //method=POST的处理
    Delete()                        //method=DELETE的处理
    Put()                           //method=PUT的处理
    Head()                          //method=HEAD的处理
    Patch()                         //method=PATCH的处理
    Options()                       //method=OPTIONS的处理
    Finish()                        //执行完成之后的处理
    Render() error                  //执行完method对应的方法之后渲染页面
}
```

那么前面介绍的路由add函数的时候是定义了ControllerInterface类型，因此，只要我们实现这个接口就可以，所以我们的基类Controller实现如下的方法：

```
func (c *Controller) Init(ct *Context, cn string) {
    c.Data = make(map[interface{}]interface{})
    c.Layout = make([]string, 0)
    c.TplNames = ""
    c.ChildName = cn
    c.Ct = ct
    c.TplExt = "tpl"
}

func (c *Controller) Prepare() {

}

func (c *Controller) Finish() {

}

func (c *Controller) Get() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Post() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
```

```
}

func (c *Controller) Delete() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Put() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Head() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Patch() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Options() {
    http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Render() error {
    if len(c.Layout) > 0 {
        var filenames []string
        for _, file := range c.Layout {
            filenames = append(filenames, path.Join(ViewsPath, file
        }
        t, err := template.ParseFiles(filenames...)
        if err != nil {
            Trace("template ParseFiles err:", err)
        }
        err = t.ExecuteTemplate(c.Ct.ResponseWriter, c.TplNames, c.
        if err != nil {
            Trace("template Execute err:", err)
        }
    } else {
        if c.TplNames == "" {
            c.TplNames = c.ChildName + "/" + c.Ct.Request.Method +
        }
        t, err := template.ParseFiles(path.Join(ViewsPath, c.TplNar
        if err != nil {
            Trace("template ParseFiles err:", err)
        }
        err = t.Execute(c.Ct.ResponseWriter, c.Data)
        if err != nil {
            Trace("template Execute err:", err)
        }
    }
    return nil
}

func (c *Controller) Redirect(url string, code int) {
```

```
    c.Ct.Redirect(code, url)
}
```

上面的controller基类已经实现了接口定义的函数，通过路由根据url执行相应的controller的原则，会依次执行如下：

```
Init()      初始化
Prepare()   执行之前的初始化，每个继承的子类可以来实现该函数
method()    根据不同的method执行不同的函数：GET、POST、PUT、HEAD等，子类来
Render()    可选，根据全局变量AutoRender来判断是否执行
Finish()    执行完之后执行的操作，每个继承的子类可以来实现该函数
```

应用指南

上面beego框架中完成了controller基类的设计，那么我们在我们的应用中可以这样来设计我们的方法：

```
package controllers

import (
    "github.com/astaxie/beego"
)

type MainController struct {
    beego.Controller
}

func (this *MainController) Get() {
    this.Data["Username"] = "astaxie"
    this.Data["Email"] = "astaxie@gmail.com"
    this.TplNames = "index.tpl"
}
```

上面的方式我们实现了子类MainController，实现了Get方法，那么如果用户通过其他的方式(POST/HEAD等)来访问该资源都将返回403，而如果是Get来访问，因为我们设置了AutoRender=true，那么在执行完Get方法之后会自动执行Render函数，就会显示如下界面：

index.tpl的代码如下所示，我们可以看到数据的设置和显示都是相当的简单方便：

```
<!DOCTYPE html>
<html>
  <head>
    <title>beego welcome template</title>
  </head>
  <body>
    <h1>Hello, world!{{.Username}},{{.Email}}</h1>
  </body>
</html>
```

links

- 目录
- 上一章: 自定义路由器设计
- 下一节: 日志和配置设计

13.4 日志和配置设计

日志和配置的重要性

前面已经介绍过日志在我们程序开发中起着很重要的作用，通过日志我们可以记录调试我们的信息，当初介绍过一个日志系统seelog，根据不同的level输出不同的日志，这个对于程序开发和程序部署来说至关重要。我们可以在程序开发中设置level低一点，部署的时候把level设置高，这样我们开发中的调试信息可以屏蔽掉。

配置模块对于应用部署牵涉到服务器不同的一些配置信息非常有用，例如一些数据库配置信息、监听端口、监听地址等都是可以通过配置文件来配置，这样我们的应用程序就具有很强的灵活性，可以通过配置文件的配置部署在不同的机器上，可以连接不同的数据库之类的。

beego的日志设计

beego的日志设计部署思路来自于seelog，根据不同的level来记录日志，但是beego设计的日志系统比较轻量级，采用了系统的log.Logger接口，默认输出到os.Stdout,用户可以实现这个接口然后通过beego.SetLogger设置自定义的输出，详细的实现如下所示：

```
// Log levels to control the logging output.
const (
    LevelTrace = iota
    LevelDebug
    LevelInfo
    LevelWarning
    LevelError
    LevelCritical
)

// logLevel controls the global log level used by the logger.
var level = LevelTrace

// LogLevel returns the global log level and can be used in
// own implementations of the logger interface.
func Level() int {
    return level
}

// SetLogLevel sets the global log level used by the simple
// logger.
func SetLevel(l int) {
    level = l
}
```

上面这一段实现了日志系统的日志分级，默认的级别是Trace，用户通过SetLevel可以设置不同的分级。

```
// logger references the used application logger.
var BeeLogger = log.New(os.Stdout, "", log.Ldate|log.Ltime)

// SetLogger sets a new logger.
func SetLogger(l *log.Logger) {
    BeeLogger = l
}

// Trace logs a message at trace level.
func Trace(v ...interface{}) {
    if level <= LevelTrace {
        BeeLogger.Printf("[T] %v\n", v)
    }
}

// Debug logs a message at debug level.
func Debug(v ...interface{}) {
    if level <= LevelDebug {
        BeeLogger.Printf("[D] %v\n", v)
    }
}

// Info logs a message at info level.
func Info(v ...interface{}) {
    if level <= LevelInfo {
        BeeLogger.Printf("[I] %v\n", v)
    }
}

// Warning logs a message at warning level.
func Warn(v ...interface{}) {
    if level <= LevelWarning {
        BeeLogger.Printf("[W] %v\n", v)
    }
}

// Error logs a message at error level.
func Error(v ...interface{}) {
    if level <= LevelError {
        BeeLogger.Printf("[E] %v\n", v)
    }
}

// Critical logs a message at critical level.
func Critical(v ...interface{}) {
    if level <= LevelCritical {
        BeeLogger.Printf("[C] %v\n", v)
    }
}
```

上面这一段代码默认初始化了一个BeeLogger对象，默认输出到os.Stdout，用户可以通过beego.SetLogger来设置实现了logger的接口输出。这里面实现了六个函数：

- Trace（一般的记录信息，举例如下：）
  - "Entered parse function validation block"
  - "Validation: entered second 'if'"
  - "Dictionary 'Dict' is empty. Using default value"
- Debug（调试信息，举例如下：）
  - "Web page requested: http://somesite.com Params='...'"
  - "Response generated. Response size: 10000. Sending."
  - "New file received. Type:PNG Size:20000"
- Info（打印信息，举例如下：）
  - "Web server restarted"
  - "Hourly statistics: Requested pages: 12345 Errors: 123 ..."
  - "Service paused. Waiting for 'resume' call"
- Warn（警告信息，举例如下：）
  - "Cache corrupted for file='test.file'. Reading from back-end"
  - "Database 192.168.0.7/DB not responding. Using backup 192.168.0.8/DB"
  - "No response from statistics server. Statistics not sent"
- Error（错误信息，举例如下：）
  - "Internal error. Cannot process request #12345 Error:...."
  - "Cannot perform login: credentials DB not responding"
- Critical（致命错误，举例如下：）
  - "Critical panic received: .... Shutting down"
  - "Fatal error: ... App is shutting down to prevent data corruption or loss"

可以看到每个函数里面都有对level的判断，所以如果我们在部署的时候设置了level=LevelWarning，那么Trace、Debug、Info这三个函数都不会有任何的输出，以此类推。

beego的配置设计

配置信息的解析，beego实现了一个key=value的配置文件读取，类似ini配置文件的格式，就是一个文件解析的过程，然后把解析的数据保存到map中，最后在调用的时候通过几个string、int之类的函数调用返回相应的值，具体的实现请看下面：

首先定义了一些ini配置文件的一些全局性常量 ：

```
var (
    bComment = []byte{'#'}
    bEmpty   = []byte{}
    bEqual   = []byte{'='}
    bDQuote  = []byte{'"'}
)
```

定义了配置文件的格式：

```
// A Config represents the configuration.
type Config struct {
    filename string
    comment  map[int][]string  // id: []{comment, key...}; id 1 is
    data     map[string]string // key: value
    offset   map[string]int64  // key: offset; for editing.
    sync.RWMutex
}
```

定义了解析文件的函数，解析文件的过程是打开文件，然后一行一行的读取，解析注释、空行和key=value数据：

```
// ParseFile creates a new Config and parses the file configuratior
// named file.
func LoadConfig(name string) (*Config, error) {
    file, err := os.Open(name)
    if err != nil {
        return nil, err
    }

    cfg := &Config{
        file.Name(),
        make(map[int][]string),
        make(map[string]string),
        make(map[string]int64),
        sync.RWMutex{},
    }
    cfg.Lock()
    defer cfg.Unlock()
    defer file.Close()

    var comment bytes.Buffer
    buf := bufio.NewReader(file)

    for nComment, off := 0, int64(1); ; {
        line, _, err := buf.ReadLine()
        if err == io.EOF {
            break
        }
        if bytes.Equal(line, bEmpty) {
            continue
        }

        off += int64(len(line))

        if bytes.HasPrefix(line, bComment) {
            line = bytes.TrimLeft(line, "#")
            line = bytes.TrimLeftFunc(line, unicode.IsSpace)
```

```
                comment.Write(line)
                comment.WriteByte('\n')
                continue
            }
            if comment.Len() != 0 {
                cfg.comment[nComment] = []string{comment.String()}
                comment.Reset()
                nComment++
            }

            val := bytes.SplitN(line, bEqual, 2)
            if bytes.HasPrefix(val[1], bDQuote) {
                val[1] = bytes.Trim(val[1], `"`)
            }

            key := strings.TrimSpace(string(val[0]))
            cfg.comment[nComment-1] = append(cfg.comment[nComment-1],
            cfg.data[key] = strings.TrimSpace(string(val[1]))
            cfg.offset[key] = off
        }
        return cfg, nil
    }
```

下面实现了一些读取配置文件的函数，返回的值确定为bool、int、float64或string：

```
// Bool returns the boolean value for a given key.
func (c *Config) Bool(key string) (bool, error) {
    return strconv.ParseBool(c.data[key])
}

// Int returns the integer value for a given key.
func (c *Config) Int(key string) (int, error) {
    return strconv.Atoi(c.data[key])
}

// Float returns the float value for a given key.
func (c *Config) Float(key string) (float64, error) {
    return strconv.ParseFloat(c.data[key], 64)
}

// String returns the string value for a given key.
func (c *Config) String(key string) string {
    return c.data[key]
}
```

应用指南

下面这个函数是我一个应用中的例子，用来获取远程url地址的json数据，实现如下：

```
func GetJson() {
    resp, err := http.Get(beego.AppConfig.String("url"))
    if err != nil {
        beego.Critical("http get info error")
        return
    }
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    err = json.Unmarshal(body, &AllInfo)
    if err != nil {
        beego.Critical("error:", err)
    }
}
```

函数中调用了框架的日志函数 `beego.Critical` 函数用来报错，调用了 `beego.AppConfig.String("url")` 用来获取配置文件中的信息，配置文件的信息如下(app.conf)：

```
appname = hs
url ="http://www.api.com/api.html"
```

links

- 目录
- 上一章: controller设计
- 下一节: 实现博客的增删改

13.5 实现博客的增删改

前面介绍了beego框架实现的整体构思以及部分实现的伪代码，这小节介绍通过beego建立一个博客系统，包括博客浏览、添加、修改、删除等操作。

博客目录

博客目录如下所示：

```
.
├── controllers
│   ├── delete.go
│   ├── edit.go
│   ├── index.go
│   ├── new.go
│   └── view.go
├── main.go
├── models
│   └── model.go
└── views
    ├── edit.tpl
    ├── index.tpl
    ├── layout.tpl
    ├── new.tpl
    └── view.tpl
```

博客路由

博客主要的路由规则如下所示：

```
//显示博客首页
beego.Router("/", &controllers.IndexController{})
//查看博客详细信息
beego.Router("/view/:id([0-9]+)", &controllers.ViewController{})
//新建博客博文
beego.Router("/new", &controllers.NewController{})
//删除博文
beego.Router("/delete/:id([0-9]+)", &controllers.DeleteController{]
//编辑博文
beego.Router("/edit/:id([0-9]+)", &controllers.EditController{})
```

数据库结构

数据库设计最简单的博客信息

```
CREATE TABLE entries (
    id INT AUTO_INCREMENT,
    title TEXT,
    content TEXT,
    created DATETIME,
    primary key (id)
);
```

控制器

IndexController:

```
type IndexController struct {
    beego.Controller
}

func (this *IndexController) Get() {
    this.Data["blogs"] = models.GetAll()
    this.Layout = "layout.tpl"
    this.TplNames = "index.tpl"
}
```

ViewController:

```
type ViewController struct {
    beego.Controller
}

func (this *ViewController) Get() {
    id, _ := strconv.Atoi(this.Ctx.Input.Params[":id"])
    this.Data["Post"] = models.GetBlog(id)
    this.Layout = "layout.tpl"
    this.TplNames = "view.tpl"
}
```

NewController

```
type NewController struct {
    beego.Controller
}

func (this *NewController) Get() {
    this.Layout = "layout.tpl"
    this.TplNames = "new.tpl"
}

func (this *NewController) Post() {
    inputs := this.Input()
    var blog models.Blog
    blog.Title = inputs.Get("title")
    blog.Content = inputs.Get("content")
    blog.Created = time.Now()
    models.SaveBlog(blog)
    this.Ctx.Redirect(302, "/")
}
```

EditController

```
type EditController struct {
    beego.Controller
}

func (this *EditController) Get() {
    id, _ := strconv.Atoi(this.Ctx.Input.Params[":id"])
    this.Data["Post"] = models.GetBlog(id)
    this.Layout = "layout.tpl"
    this.TplNames = "edit.tpl"
}

func (this *EditController) Post() {
    inputs := this.Input()
    var blog models.Blog
    blog.Id, _ = strconv.Atoi(inputs.Get("id"))
    blog.Title = inputs.Get("title")
    blog.Content = inputs.Get("content")
    blog.Created = time.Now()
    models.SaveBlog(blog)
    this.Ctx.Redirect(302, "/")
}
```

DeleteController

```
type DeleteController struct {
    beego.Controller
}

func (this *DeleteController) Get() {
    id, _ := strconv.Atoi(this.Ctx.Input.Params[":id"])
    blog := models.GetBlog(id)
    this.Data["Post"] = blog
    models.DelBlog(blog)
    this.Ctx.Redirect(302, "/")
}
```

model层

```
package models

import (
    "database/sql"
    "github.com/astaxie/beedb"
    _ "github.com/ziutek/mymysql/godrv"
    "time"
)

type Blog struct {
    Id      int `PK`
    Title   string
    Content string
    Created time.Time
}

func GetLink() beedb.Model {
    db, err := sql.Open("mymysql", "blog/astaxie/123456")
    if err != nil {
        panic(err)
    }
    orm := beedb.New(db)
    return orm
}

func GetAll() (blogs []Blog) {
    db := GetLink()
    db.FindAll(&blogs)
    return
}

func GetBlog(id int) (blog Blog) {
    db := GetLink()
    db.Where("id=?", id).Find(&blog)
    return
}

func SaveBlog(blog Blog) (bg Blog) {
    db := GetLink()
    db.Save(&blog)
    return bg
}

func DelBlog(blog Blog) {
    db := GetLink()
    db.Delete(&blog)
    return
}
```

view层

layout.tpl

```
<html>
<head>
    <title>My Blog</title>
    <style>
        #menu {
            width: 200px;
            float: right;
        }
    </style>
</head>
<body>

<ul id="menu">
    <li><a href="/">Home</a></li>
    <li><a href="/new">New Post</a></li>
</ul>

{{.LayoutContent}}

</body>
</html>
```

index.tpl

```
<h1>Blog posts</h1>

<ul>
{{range .blogs}}
    <li>
        <a href="/view/{{.Id}}">{{.Title}}</a>
        from {{.Created}}
        <a href="/edit/{{.Id}}">Edit</a>
        <a href="/delete/{{.Id}}">Delete</a>
    </li>
{{end}}
</ul>
```

view.tpl

```
<h1>{{.Post.Title}}</h1>
{{.Post.Created}}<br/>

{{.Post.Content}}
```

new.tpl

```
<h1>New Blog Post</h1>
<form action="" method="post">
标题:<input type="text" name="title"><br>
内容：<textarea name="content" colspan="3" rowspan="10"></textarea>
<input type="submit">
</form>
```

edit.tpl

```
<h1>Edit {{.Post.Title}}</h1>

<h1>New Blog Post</h1>
<form action="" method="post">
标题:<input type="text" name="title" value="{{.Post.Title}}"><br>
内容：<textarea name="content" colspan="3" rowspan="10">{{.Post.Cont
<input type="hidden" name="id" value="{{.Post.Id}}">
<input type="submit">
</form>
```

links

- 目录
- 上一章: 日志和配置设计
- 下一节: 小结

13.6 小结

这一章我们主要介绍了如何实现一个基础的Go语言框架，框架包含有路由设计，由于Go内置的http包中路由的一些不足点，我们设计了动态路由规则，然后介绍了MVC模式中的Controller设计，controller实现了REST的实现，这个主要思路来源于tornado框架，然后设计实现了模板的layout以及自动化渲染等技术，主要采用了Go内置的模板引擎，最后我们介绍了一些辅助的日志、配置等信息的设计，通过这些设计我们实现了一个基础的框架beego，目前该框架已经开源在github，最后我们通过beego实现了一个博客系统，通过实例代码详细的展现了如何快速的开发一个站点。

links

- 目录
- 上一章: 实现博客的增删改
- 下一节: 扩展Web框架

14 扩展Web框架

第十三章介绍了如何开发一个Web框架，通过介绍MVC、路由、日志处理、配置处理完成了一个基本的框架系统，但是一个好的框架需要一些方便的辅助工具来快速的开发Web，那么我们这一章将就如何提供一些快速开发Web的工具进行介绍，第一小节介绍如何处理静态文件，如何利用现有的twitter开源的bootstrap进行快速的开发美观的站点，第二小节介绍如何利用前面介绍的session来进行用户登录处理，第三小节介绍如何方便的输出表单、这些表单如何进行数据验证，如何快速的结合model进行数据的增删改操作，第四小节介绍如何进行一些用户认证，包括http basic认证、http digest认证，第五小节介绍如何利用前面介绍的i18n支持多语言的应用开发。第六小节介绍了如何集成Go的pprof包用于性能调试。

通过本章的扩展，beego框架将具有快速开发Web的特性，最后我们将讲解如何利用这些扩展的特性扩展开发第十三章开发的博客系统，通过开发一个完整、美观的博客系统让读者了解beego开发带给你的快速。

目录

links

- 目录
- 上一章: 第十三章总结
- 下一节: 静态文件支持

14.1 静态文件支持

我们在前面已经讲过如何处理静态文件，这小节我们详细的介绍如何在beego里面设置和使用静态文件。通过再介绍一个twitter开源的html、css框架bootstrap，无需大量的设计工作就能够让你快速地建立一个漂亮的站点。

beego静态文件实现和设置

Go的net/http包中提供了静态文件的服务， `ServeFile` 和 `FileServer` 等函数。beego的静态文件处理就是基于这一层处理的，具体的实现如下所示：

```
//static file server
for prefix, staticDir := range StaticDir {
    if strings.HasPrefix(r.URL.Path, prefix) {
        file := staticDir + r.URL.Path[len(prefix):]
        http.ServeFile(w, r, file)
        w.started = true
        return
    }
}
```

StaticDir里面保存的是相应的url对应到静态文件所在的目录，因此在处理URL请求的时候只需要判断对应的请求地址是否包含静态处理开头的url，如果包含的话就采用http.ServeFile提供服务。

举例如下：

```
beego.StaticDir["/asset"] = "/static"
```

那么请求url如 `http://www.beego.me/asset/bootstrap.css` 就会请求 `/static/bootstrap.css` 来提供反馈给客户端。

bootstrap集成

Bootstrap是Twitter推出的一个开源的用于前端开发的工具包。对于开发者来说，Bootstrap是快速开发Web应用程序的最佳前端工具包。它是一个CSS和HTML的集合，它使用了最新的HTML5标准，给你的Web开发提供了时尚的版式，表单，按钮，表格，网格系统等等。

- 组件　　Bootstrap中包含了丰富的Web组件，根据这些组件，可以快速的搭建一个漂亮、功能完备的网站。其中包括以下组件：　　下拉菜单、按钮组、按钮下拉菜单、导航、导航条、面包屑、分页、排版、缩略图、警告对话框、

进度条、媒体对象等
- Javascript插件　　Bootstrap自带了13个jQuery插件，这些插件为Bootstrap中的组件赋予了“生命”。其中包括：　　模式对话框、标签页、滚动条、弹出框等。
- 定制自己的框架代码　　可以对Bootstrap中所有的CSS变量进行修改，依据自己的需求裁剪代码。

图14.1 bootstrap站点

接下来我们利用bootstrap集成到beego框架里面来，快速的建立一个漂亮的站点。

1. 首先把下载的bootstrap目录放到我们的项目目录，取名为static，如下截图所示

   图14.2 项目中静态文件目录结构

2. 因为beego默认设置了StaticDir的值，所以如果你的静态文件目录是static的话就无须再增加了：

   StaticDir["/static"] = "static"

3. 模板中使用如下的地址就可以了：

```
//css文件
<link href="/static/css/bootstrap.css" rel="stylesheet">

//js文件
<script src="/static/js/bootstrap-transition.js"></script>

//图片文件
<img src="/static/img/logo.png">
```

上面可以实现把bootstrap集成到beego中来，如下展示的图就是集成进来之后的展现效果图：

图14.3 构建的基于bootstrap的站点界面

这些模板和格式bootstrap官方都有提供，这边就不再重复贴代码，大家可以上bootstrap官方网站学习如何编写模板。

links

- 目录
- 上一节: 扩展Web框架

- 下一节: Session支持

14.2 Session支持

第六章的时候我们介绍过如何在Go语言中使用session，也实现了一个sessionManger，beego框架基于sessionManager实现了方便的session处理功能。

session集成

beego中主要有以下的全局变量来控制session处理：

```
//related to session
SessionOn            bool   // 是否开启session模块，默认不开启
SessionProvider      string // session后端提供处理模块，默认是sessionMa
SessionName          string // 客户端保存的cookies的名称
SessionGCMaxLifetime int64  // cookies有效期

GlobalSessions *session.Manager //全局session控制器
```

当然上面这些变量需要初始化值，也可以按照下面的代码来配合配置文件以设置这些值：

```
if ar, err := AppConfig.Bool("sessionon"); err != nil {
    SessionOn = false
} else {
    SessionOn = ar
}
if ar := AppConfig.String("sessionprovider"); ar == "" {
    SessionProvider = "memory"
} else {
    SessionProvider = ar
}
if ar := AppConfig.String("sessionname"); ar == "" {
    SessionName = "beegosessionID"
} else {
    SessionName = ar
}
if ar, err := AppConfig.Int("sessiongcmaxlifetime"); err != nil &&
    int64val, _ := strconv.ParseInt(strconv.Itoa(ar), 10, 64)
    SessionGCMaxLifetime = int64val
} else {
    SessionGCMaxLifetime = 3600
}
```

在beego.Run函数中增加如下代码：

```
if SessionOn {
    GlobalSessions, _ = session.NewManager(SessionProvider, Sessior
    go GlobalSessions.GC()
}
```

这样只要SessionOn设置为true，那么就会默认开启session功能，独立开一个goroutine来处理session。

为了方便我们在自定义Controller中快速使用session，作者在`beego.Controller`中提供了如下方法：

```
func (c *Controller) StartSession() (sess session.Session) {
    sess = GlobalSessions.SessionStart(c.Ctx.ResponseWriter, c.Ctx
    return
}
```

session使用

通过上面的代码我们可以看到，beego框架简单地继承了session功能，那么在项目中如何使用呢？

首先我们需要在应用的main入口处开启session：

```
beego.SessionOn = true
```

然后我们就可以在控制器的相应方法中如下所示的使用session了：

```
func (this *MainController) Get() {
    var intcount int
    sess := this.StartSession()
    count := sess.Get("count")
    if count == nil {
        intcount = 0
    } else {
        intcount = count.(int)
    }
    intcount = intcount + 1
    sess.Set("count", intcount)
    this.Data["Username"] = "astaxie"
    this.Data["Email"] = "astaxie@gmail.com"
    this.Data["Count"] = intcount
    this.TplNames = "index.tpl"
}
```

上面的代码展示了如何在控制逻辑中使用session，主要分两个步骤：

1. 获取session对象

```
//获取对象,类似PHP中的session_start()
sess := this.StartSession()
```

2. 使用session进行一般的session值操作

```
//获取session值，类似PHP中的$_SESSION["count"]
sess.Get("count")

//设置session值
sess.Set("count", intcount)
```

从上面代码可以看出基于beego框架开发的应用中使用session相当方便，基本上和PHP中调用 `session_start()` 类似。

links

- 目录
- 上一节: 静态文件支持
- 下一节: 表单及验证支持

14.3 表单及验证支持

在Web开发中对于这样的一个流程可能很眼熟：

- 打开一个网页显示出表单。
- 用户填写并提交了表单。
- 如果用户提交了一些无效的信息，或者可能漏掉了一个必填项，表单将会连同用户的数据和错误问题的描述信息返回。
- 用户再次填写，继续上一步过程，直到提交了一个有效的表单。

在接收端，脚本必须：

- 检查用户递交的表单数据。
- 验证数据是否为正确的类型，合适的标准。例如，如果一个用户名被提交，它必须被验证是否只包含了允许的字符。它必须有一个最小长度，不能超过最大长度。用户名不能与已存在的他人用户名重复，甚至是一个保留字等。
- 过滤数据并清理不安全字符，保证逻辑处理中接收的数据是安全的。
- 如果需要，预格式化数据（数据需要清除空白或者经过HTML编码等等。）
- 准备好数据，插入数据库。

尽管上面的过程并不是很复杂，但是通常情况下需要编写很多代码，而且为了显示错误信息，在网页中经常要使用多种不同的控制结构。创建表单验证虽简单，实施起来实在枯燥无味。

表单和验证

对于开发者来说，一般开发过程都是相当复杂，而且大多是在重复一样的工作。假设一个场景项目中忽然需要增加一个表单数据，那么局部代码的整个流程都需要修改。我们知道Go里面struct是常用的一个数据结构，因此beego的form采用了struct来处理表单信息。

首先定义一个开发Web应用时相对应的struct，一个字段对应一个form元素，通过struct的tag来定义相应的元素信息和验证信息，如下所示：

```
type User struct{
    Username     string      `form:text,valid:required`
    Nickname     string      `form:text,valid:required`
    Age            int      `form:text,valid:required|numeric`
    Email         string      `form:text,valid:required|valid_email
    Introduce     string      `form:textarea`
}
```

定义好struct之后接下来在controller中这样操作

```
func (this *AddController) Get() {
    this.Data["form"] = beego.Form(&User{})
    this.Layout = "admin/layout.html"
    this.TplNames = "admin/add.tpl"
}
```

在模板中这样显示表单

```
<h1>New Blog Post</h1>
<form action="" method="post">
{{.form.render()}}
</form>
```

上面我们定义好了整个的第一步，从struct到显示表单的过程，接下来就是用户填写信息，服务器端接收数据然后验证，最后插入数据库。

```
func (this *AddController) Post() {
    var user User
    form := this.GetInput(&user)
    if !form.Validates() {
        return
    }
    models.UserInsert(&user)
    this.Ctx.Redirect(302, "/admin/index")
}
```

表单类型

以下列表列出来了对应的form元素信息：

名称	参数	功能描述
text	No	textbox输入框
button	No	按钮
checkbox	No	多选择框
dropdown	No	下拉选择框
file	No	文件上传
hidden	No	隐藏元素
password	No	密码输入框
radio	No	单选框
textarea	No	文本输入框

表单验证

以下列表将列出可被使用的原生规则

规则	参数	描述	举例
required	No	如果元素为空，则返回FALSE	
matches	Yes	如果表单元素的值与参数中对应的表单字段的值不相等，则返回FALSE	matches[form_item]
is_unique	Yes	如果表单元素的值与指定数据表栏位有重复，则返回False（译者注：比如is_unique[User.Email]，那么验证类会去查找User表中Email栏位有没有与表单元素一样的值，如存重复，则返回false，这样开发者就不必另写Callback验证代码。）	is_unique[table.field]
min_length	Yes	如果表单元素值的字符长度少于参数中定义的数字，则返回FALSE	min_length[6]

max_length	Yes	如果表单元素值的字符长度大于参数中定义的数字，则返回FALSE	max_length[12]
exact_length	Yes	如果表单元素值的字符长度与参数中定义的数字不符，则返回FALSE	exact_length[8]
greater_than	Yes	如果表单元素值是非数字类型，或小于参数定义的值，则返回FALSE	greater_than[8]
less_than	Yes	如果表单元素值是非数字类型，或大于参数定义的值，则返回FALSE	less_than[8]
alpha	No	如果表单元素值中包含除字母以外的其他字符，则返回FALSE	
alpha_numeric	No	如果表单元素值中包含除字母和数字以外的其他字符，则返回FALSE	
alpha_dash	No	如果表单元素值中包含除字母/数字/下划线/破折号以外的其他字符，则返回FALSE	
numeric	No	如果表单元素值中包含除数字以外的字符，则返回 FALSE	
integer	No	如果表单元素中包含除整数以外的字符，则返回FALSE	
decimal	Yes	如果表单元素中输入（非小数）不完整的值，则返回FALSE	
is_natural	No	如果表单元素值中包含了非自然数的其他数值（其他数值不包括零），则返回FALSE。自然数形如：0,1,2,3....等等。	
is_natural_no_zero	No	如果表单元素值包含了非自然数的其他数值（其他数值包括零），则返回FALSE。非零的	

		自然数：1,2,3.....等等。	
valid_email	No	如果表单元素值包含不合法的email地址，则返回FALSE	
valid_emails	No	如果表单元素值中任何一个值包含不合法的email地址（地址之间用英文逗号分割），则返回FALSE。	
valid_ip	No	如果表单元素的值不是一个合法的IP地址，则返回FALSE。	
valid_base64	No	如果表单元素的值包含除了base64 编码字符之外的其他字符，则返回FALSE。	

links

- 目录
- 上一节: Session支持
- 下一节: 用户认证

14.4 用户认证

在开发Web应用过程中，用户认证是开发者经常遇到的问题，用户登录、注册、登出等操作，而一般认证也分为三个方面的认证

- HTTP Basic和 HTTP Digest认证
- 第三方集成认证：QQ、微博、豆瓣、OPENID、google、github、facebook和twitter等
- 自定义的用户登录、注册、登出，一般都是基于session、cookie认证

beego目前没有针对这三种方式进行任何形式的集成，但是可以充分的利用第三方开源库来实现上面的三种方式的用户认证，不过后续beego会对前面两种认证逐步集成。

HTTP Basic和 HTTP Digest认证

这两个认证是一些应用采用的比较简单的认证，目前已经有开源的第三方库支持这两个认证：

```
github.com/abbot/go-http-auth
```

下面代码演示了如何把这个库引入beego中从而实现认证：

```
package controllers

import (
    "github.com/abbot/go-http-auth"
    "github.com/astaxie/beego"
)

func Secret(user, realm string) string {
    if user == "john" {
        // password is "hello"
        return "$1$dlPL2MqE$oQmn16q49SqdmhenQuNgs1"
    }
    return ""
}

type MainController struct {
    beego.Controller
}

func (this *MainController) Prepare() {
    a := auth.NewBasicAuthenticator("example.com", Secret)
    if username := a.CheckAuth(this.Ctx.Request); username == "" {
        a.RequireAuth(this.Ctx.ResponseWriter, this.Ctx.Request)
    }
}

func (this *MainController) Get() {
    this.Data["Username"] = "astaxie"
    this.Data["Email"] = "astaxie@gmail.com"
    this.TplNames = "index.tpl"
}
```

上面代码利用了beego的prepare函数，在执行正常逻辑之前调用了认证函数，这样就非常简单的实现了http auth，digest的认证也是同样的原理。

oauth和oauth2的认证

oauth和oauth2是目前比较流行的两种认证方式，还好第三方有一个库实现了这个认证，但是是国外实现的，并没有QQ、微博之类的国内应用认证集成：

```
github.com/bradrydzewski/go.auth
```

下面代码演示了如何把该库引入beego中从而实现oauth的认证，这里以github为例演示：

1. 添加两条路由

```
beego.RegisterController("/auth/login", &controllers.GithubCon
beego.RegisterController("/mainpage", &controllers.PageControl
```

2. 然后我们处理GithubController登陆的页面：

```
package controllers

import (
    "github.com/astaxie/beego"
    "github.com/bradrydzewski/go.auth"
)

const (
    githubClientKey = "a0864ea791ce7e7bd0df"
    githubSecretKey = "a0ec09a647a688a64a28f6190b5a0d2705df56c
)

type GithubController struct {
    beego.Controller
}

func (this *GithubController) Get() {
    // set the auth parameters
    auth.Config.CookieSecret = []byte("7H9xiimk2QdTdYI7rDddfJe
    auth.Config.LoginSuccessRedirect = "/mainpage"
    auth.Config.CookieSecure = false

    githubHandler := auth.Github(githubClientKey, githubSecret

    githubHandler.ServeHTTP(this.Ctx.ResponseWriter, this.Ctx.
}
```

1. 处理登陆成功之后的页面

```
package controllers

import (
    "github.com/astaxie/beego"
    "github.com/bradrydzewski/go.auth"
    "net/http"
    "net/url"
)

type PageController struct {
    beego.Controller
}

func (this *PageController) Get() {
    // set the auth parameters
    auth.Config.CookieSecret = []byte("7H9xiimk2QdTdYI7rDddfJe
    auth.Config.LoginSuccessRedirect = "/mainpage"
    auth.Config.CookieSecure = false

    user, err := auth.GetUserCookie(this.Ctx.Request)

    //if no active user session then authorize user
    if err != nil || user.Id() == "" {
        http.Redirect(this.Ctx.ResponseWriter, this.Ctx.Reques
        return
    }

    //else, add the user to the URL and continue
    this.Ctx.Request.URL.User = url.User(user.Id())
    this.Data["pic"] = user.Picture()
    this.Data["id"] = user.Id()
    this.Data["name"] = user.Name()
    this.TplNames = "home.tpl"
}
```

整个的流程如下，首先打开浏览器输入地址：

图14.4 显示带有登录按钮的首页

然后点击链接出现如下界面：

图14.5 点击登录按钮后显示github的授权页

然后点击Authorize app就出现如下界面：

图14.6 授权登录之后显示的获取到的github信息页

自定义认证

自定义的认证一般都是和session结合验证的，如下代码来源于一个基于beego的开源博客：

```
//登陆处理
func (this *LoginController) Post() {
    this.TplNames = "login.tpl"
    this.Ctx.Request.ParseForm()
    username := this.Ctx.Request.Form.Get("username")
    password := this.Ctx.Request.Form.Get("password")
    md5Password := md5.New()
    io.WriteString(md5Password, password)
    buffer := bytes.NewBuffer(nil)
    fmt.Fprintf(buffer, "%x", md5Password.Sum(nil))
    newPass := buffer.String()

    now := time.Now().Format("2006-01-02 15:04:05")

    userInfo := models.GetUserInfo(username)
    if userInfo.Password == newPass {
        var users models.User
        users.Last_logintime = now
        models.UpdateUserInfo(users)

        //登录成功设置session
        sess := globalSessions.SessionStart(this.Ctx.ResponseWriter
        sess.Set("uid", userInfo.Id)
        sess.Set("uname", userInfo.Username)

        this.Ctx.Redirect(302, "/")
    }
}

//注册处理
func (this *RegController) Post() {
    this.TplNames = "reg.tpl"
    this.Ctx.Request.ParseForm()
    username := this.Ctx.Request.Form.Get("username")
    password := this.Ctx.Request.Form.Get("password")
    usererr := checkUsername(username)
    fmt.Println(usererr)
    if usererr == false {
        this.Data["UsernameErr"] = "Username error, Please to again
        return
    }

    passerr := checkPassword(password)
    if passerr == false {
```

```
        this.Data["PasswordErr"] = "Password error, Please to agair
        return
    }

    md5Password := md5.New()
    io.WriteString(md5Password, password)
    buffer := bytes.NewBuffer(nil)
    fmt.Fprintf(buffer, "%x", md5Password.Sum(nil))
    newPass := buffer.String()

    now := time.Now().Format("2006-01-02 15:04:05")

    userInfo := models.GetUserInfo(username)

    if userInfo.Username == "" {
        var users models.User
        users.Username = username
        users.Password = newPass
        users.Created = now
        users.Last_logintime = now
        models.AddUser(users)

        //登录成功设置session
        sess := globalSessions.SessionStart(this.Ctx.ResponseWriter
        sess.Set("uid", userInfo.Id)
        sess.Set("uname", userInfo.Username)
        this.Ctx.Redirect(302, "/")
    } else {
        this.Data["UsernameErr"] = "User already exists"
    }

}

func checkPassword(password string) (b bool) {
    if ok, _ := regexp.MatchString("^[a-zA-Z0-9]{4,16}$", password)
        return false
    }
    return true
}

func checkUsername(username string) (b bool) {
    if ok, _ := regexp.MatchString("^[a-zA-Z0-9]{4,16}$", username)
        return false
    }
    return true
}
```

有了用户登陆和注册之后，其他模块的地方可以增加如下这样的用户是否登陆的判断：

```
func (this *AddBlogController) Prepare() {
    sess := globalSessions.SessionStart(this.Ctx.ResponseWriter, th
    sess_uid := sess.Get("userid")
    sess_username := sess.Get("username")
    if sess_uid == nil {
        this.Ctx.Redirect(302, "/admin/login")
        return
    }
    this.Data["Username"] = sess_username
}
```

links

- 目录
- 上一节: 表单及验证支持
- 下一节: 多语言支持

14.5 多语言支持

我们在第十章介绍过国际化和本地化，开发了一个go-i18n库，这小节我们将把该库集成到beego框架里面来，使得我们的框架支持国际化和本地化。

i18n集成

beego中设置全局变量如下：

```
Translation    i18n.IL
Lang          string  //设置语言包，zh、en
LangPath     string  //设置语言包所在位置
```

初始化多语言函数:

```
func InitLang(){
    beego.Translation:=i18n.NewLocale()
    beego.Translation.LoadPath(beego.LangPath)
    beego.Translation.SetLocale(beego.Lang)
}
```

为了方便在模板中直接调用多语言包，我们设计了三个函数来处理响应的多语言：

```
beegoTplFuncMap["Trans"] = i18n.I18nT
beegoTplFuncMap["TransDate"] = i18n.I18nTimeDate
beegoTplFuncMap["TransMoney"] = i18n.I18nMoney

func I18nT(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    return beego.Translation.Translate(s)
}

func I18nTimeDate(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    return beego.Translation.Time(s)
}

func I18nMoney(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }
    return beego.Translation.Money(s)
}
```

多语言开发使用

1. 设置语言以及语言包所在位置，然后初始化i18n对象：

```
beego.Lang = "zh"
beego.LangPath = "views/lang"
beego.InitLang()
```

2. 设计多语言包

   上面讲了如何初始化多语言包，现在设计多语言包，多语言包是json文件，如第十章介绍的一样，我们需要把设计的文件放在LangPath下面，例如zh.json或者en.json

   ```
   # zh.json

   {
   "zh": {
       "submit": "提交",
       "create": "创建"
       }
   }

   #en.json

   {
   "en": {
       "submit": "Submit",
       "create": "Create"
       }
   }
   ```

3. 使用语言包

   我们可以在controller中调用翻译获取响应的翻译语言，如下所示：

   ```
   func (this *MainController) Get() {
       this.Data["create"] = beego.Translation.Translate("create"
       this.TplNames = "index.tpl"
   }
   ```

   我们也可以在模板中直接调用响应的翻译函数：

   ```
   //直接文本翻译
   {{.create | Trans}}

   //时间翻译
   {{.time | TransDate}}

   //货币翻译
   {{.money | TransMoney}}
   ```

links

- 目录
- 上一节: 用户认证
- 下一节: pprof支持

14.6 pprof支持

Go语言有一个非常棒的设计就是标准库里面带有代码的性能监控工具，在两个地方有包：

```
net/http/pprof

runtime/pprof
```

其实net/http/pprof中只是使用runtime/pprof包来进行封装了一下，并在http端口上暴露出来

beego支持pprof

目前beego框架新增了pprof，该特性默认是不开启的，如果你需要测试性能，查看相应的执行goroutine之类的信息，其实Go的默认包"net/http/pprof"已经具有该功能，如果按照Go默认的方式执行Web，默认就可以使用，但是由于beego重新封装了ServHTTP函数，默认的包是无法开启该功能的，所以需要对beego的内部改造支持pprof。

- 首先在beego.Run函数中根据变量是否自动加载性能包

```
if PprofOn {
    BeeApp.RegisterController(`/debug/pprof`, &ProfController
    BeeApp.RegisterController(`/debug/pprof/:pp([\w]+)`, &Prc
}
```

- 设计ProfConterller

```
package beego

import (
    "net/http/pprof"
)

type ProfController struct {
    Controller
}

func (this *ProfController) Get() {
    switch this.Ctx.Params[":pp"] {
    default:
        pprof.Index(this.Ctx.ResponseWriter, this.Ctx.Request
    case "":
        pprof.Index(this.Ctx.ResponseWriter, this.Ctx.Request
    case "cmdline":
        pprof.Cmdline(this.Ctx.ResponseWriter, this.Ctx.Reque
    case "profile":
        pprof.Profile(this.Ctx.ResponseWriter, this.Ctx.Reque
    case "symbol":
        pprof.Symbol(this.Ctx.ResponseWriter, this.Ctx.Reques
    }
    this.Ctx.ResponseWriter.WriteHeader(200)
}
```

使用入门

通过上面的设计，你可以通过如下代码开启pprof：

```
beego.PprofOn = true
```

然后你就可以在浏览器中打开如下URL就看到如下界面：

图14.7 系统当前goroutine、heap、thread信息

点击goroutine我们可以看到很多详细的信息：

图14.8 显示当前goroutine的详细信息

我们还可以通过命令行获取更多详细的信息

```
go tool pprof http://localhost:8080/debug/pprof/profile
```

这时候程序就会进入30秒的profile收集时间，在这段时间内拼命刷新浏览器上的页面，尽量让cpu占用性能产生数据。

```
(pprof) top10

Total: 3 samples

   1 33.3% 33.3% 1 33.3% MHeap_AllocLocked

   1 33.3% 66.7% 1 33.3% os/exec.(*Cmd).closeDescriptors

   1 33.3% 100.0% 1 33.3% runtime.sigprocmask

   0 0.0% 100.0% 1 33.3% MCentral_Grow

   0 0.0% 100.0% 2 66.7% main.Compile

   0 0.0% 100.0% 2 66.7% main.compile

   0 0.0% 100.0% 2 66.7% main.run

   0 0.0% 100.0% 1 33.3% makeslice1

   0 0.0% 100.0% 2 66.7% net/http.(*ServeMux).ServeHTTP

   0 0.0% 100.0% 2 66.7% net/http.(*conn).serve

(pprof)web
```

图14.9 展示的执行流程信息

links

- 目录
- 上一节: 多语言支持
- 下一节: 小结

14.7 小结

这一章主要阐述了如何基于beego框架进行扩展，这包括静态文件的支持，静态文件主要讲述了如何利用beego进行快速的网站开发，利用bootstrap搭建漂亮的站点；第二小结讲解了如何在beego中集成sessionManager，方便用户在利用beego的时候快速的使用session；第三小结介绍了表单和验证，基于Go语言的struct的定义使得我们在开发Web的过程中从重复的工作中解放出来，而且加入了验证之后可以尽量做到数据安全，第四小结介绍了用户认证，用户认证主要有三方面的需求，http basic和http digest认证，第三方认证，自定义认证，通过代码演示了如何利用现有的第三方包集成到beego应用中来实现这些认证；第五小节介绍了多语言的支持，beego中集成了go-i18n这个多语言包，用户可以很方便的利用该库开发多语言的Web应用；第六小节介绍了如何集成Go的pprof包，pprof包是用于性能调试的工具，通过对beego的改造之后集成了pprof包，使得用户可以利用pprof测试基于beego开发的应用，通过这六个小节的介绍我们扩展出来了一个比较强壮的beego框架，这个框架足以应付目前大多数的Web应用，用户可以继续发挥自己的想象力去扩展，我这里只是简单的介绍了我能想的到的几个比较重要的扩展。

links

- 目录
- 上一节: pprof支持

附录A 参考资料

这本书的内容基本上是我学习Go过程以及以前从事Web开发过程中的一些经验总结，里面部分内容参考了很多站点的内容，感谢这些站点的内容让我能够总结出来这本书，参考资料如下：

1. golang blog
2. Russ Cox blog
3. go book
4. golangtutorials
5. 轩脉刃de刀光剑影
6. Go 官网文档
7. Network programming with Go
8. setup-the-rails-application-for-internationalization
9. The Cross-Site Scripting (XSS) FAQ
10. Network programming with Go